The Dissertation Committee for Premkishore Shivakumar
certifies that this is the approved version of the following dissertation:

Techniques to Improve the Hard and Soft Error Reliability of Distributed Architectures

Committee:

Stephen W. Keckler, Supervisor

Douglas C. Burger

Norman P. Jouppi

Calvin Lin

Steven K. Reinhardt

Nur A. Touba

Techniques to Improve the Hard and Soft Error Reliability of Distributed Architectures

by

Premkishore Shivakumar, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

To my loving wife, Simrit,

and

my loving parents.

Acknowledgments

I would like to thank and remember numerous people who have contributed to this research and my experience at UT with heartfelt gratitude. I am deeply indebted to my guide, mentor, and thesis advisor Steve Keckler. Under his guidance, I feel that I have matured tremendously in my approach to thinking about research problems and system building. I would also like to thank my co-advisor, Doug Burger. Doug gave me an opportunity to work on reliability, and much of the combinational logic soft error rate modeling work was done as part of a project in his graduate computer architecture course. I have always been inspired by Doug's quick thinking, and his ability to extract key insights from results. I cannot thank them enough for the things that I have learned from them, but I am sure what I have learned will guide me through the rest of my professional career.

I would also like to gratefully acknowledge the rest of my committee, Calvin Lin, Norm Jouppi, Steve Reinhardt, and Nur Touba, who have helped me in innumerable aspects of my research. I had the opportunity of working with Calvin on some of the initial work on hard error reliability in the TRIPS architecture, and his comments were very useful in shaping the direction of this research. I was also fortunate to work with Norm in the summer of 2001 on CACTI 3.0. I was fascinated by his wide range of expertise, and I still remember his words of advice about keeping an open mind and remembering that solutions which did not apply in an earlier context might be applicable again in a new context in the future. I am also very grateful to Steve Reinhardt, and Nur Touba for their comments which have substantially improved the quality of this dissertation.

Mike Kistler was great to work with on the soft error modeling work, and both of us gained an incredible amount of knowledge in those two years. I would have loved to continue working with him on the rest of this research. Special thanks to Heather for the help with all our SPICE related questions then! Most of this research would have not been possible without the TRIPS hardware and software infrastructure, and for that I am thankful to the entire hardware and software team. In particular, I would like to thank Ramdas, Karu, Simha, and Nitya for answering all my questions regarding TRIPS and computer architecture in general, and Katie and Xia for helping me understand the TRIPS scheduler. I was extremely fortunate to be part the TRIPS hardware design team. It was an incredible learning experience that is rarely possible in graduate school or in industry, and I had a wonderful time working closely with Nitya and Divya on the Execution Tile design. I also gained invaluable industry perspective by working with Chuck Moore on multiple things including reliability, and the TRIPS design, and have had innumerable conversations with him asking for advice regarding my professional career. Thanks Chuck!

I would also like to thank all the past and present members of the CART lab including Karu, Ramdas, Simha, Heather, CK, Sibi, Hrishi, Nitya, Haiming, Jay, Vikas, Divya, Xia, Behnam, AK, Sadia, Raj, Katie, Bert, Aaron, Hadi, Dong, Paul, and Mark who helped create a very friendly and interactive environment for useful research related and general discussions. Their feedback on my research, comments at practice talks, and feedback on paper drafts was an invaluable help to me. I have grown tremendously, both technically and otherwise, from my association with them.

I would also like to thank the friendly, patient, and efficient staff in the Computer Science department. In particular, special thanks to Gem, Gloria, and Katherine who helped me tackle the maze that is graduate school, and took care of all the deadlines, requirements, and travel arrangements.

I also made some very good friends during my stay at UT who have made graduate school an even more memorable experience for me. I could not have succeeded at graduate

school without the upbringing, encouragement, and support from my dear parents, brother, and sister. In these few years at UT, I also found two things that have deeply touched my heart. One of them is Simrit, my wife, who enriches my life beyond measure, and the other has given a deeper meaning to my existence and given my life a purpose.

PREMKISHORE SHIVAKUMAR

The University of Texas at Austin
August 2007

Techniques to Improve the Hard and Soft Error Reliability of Distributed Architectures

Publication No. _____

Premkishore Shivakumar, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Stephen W. Keckler

Aggressive technology scaling, rising on-chip integration, and the continued increase in microprocessor power and thermal density threaten both the hard and soft error reliability of future microprocessor designs. Therefore, designing low overhead mechanisms for improving reliability will be a critical requirement at future technologies. Technology constraints of wire-delay and power consumption, and limits on deep pipelining, have impelled a shift to distributed architectures that rely on modularity in design, and on-chip interconnection networks for communication, and place a greater burden on software for exploiting concurrency from the application to achieve high performance on the distributed substrate [1]. The focus of this dissertation is on architectural techniques for improving the hard and soft error

reliability of future technology-scalable distributed architectures. We make the key observation that these underlying principles of distributed architectures have important synergies that can be exploited to improve the hard and soft error reliability of microprocessors at low overhead.

Using a detailed end-to-end model for chip yield, we demonstrate that with just redundant rows and columns in memory arrays and caches the yield of chip multiprocessors drops substantially from 85% at 250nm to 60% at 50nm. We exploit the three principles of modern and future distributed architectures: the abundant microarchitectural redundancy provided by modular design, the natural redundancy in communication paths provided by multi-hop, routed, on-chip networks, and the availability of greater software assistance; for efficiently managing the redundancy to improve yield at low performance overhead. Using just modular redundancy at the intra- and inter-processor granularity, we improve the yield of chip multiprocessors to 99.6% at 50nm, with a maximum reduction in performance in any chip of less than 20%. Further, we extend this technique to take advantage of the block-atomic, and static-placement-dynamic-issue execution model in the TRIPS architecture to efficiently manage the redundancy provided by modular design and on-chip networks. Our evaluation of this compiler-assisted yield enhancement technique in the TRIPS architecture shows significant yield improvement with less than 4% impact on performance.

This dissertation also quantitatively demonstrates through detailed modeling that the raw soft error rate, especially that of combinational logic, will increase substantially at future technologies. This emphasizes the need for innovative solutions that extend soft error protection to latches, and combinational logic, while appropriately balancing the power consumption, area, and complexity overhead. We propose a new class of better-than-worst-case soft error reliability techniques called AVF throttling, that trade concurrency for reducing the amount of processor state vulnerable to soft errors. Since future architectures must increasingly rely on exploiting concurrency for achieving high performance, they aggressively bring future program state into the processor and mine them for available parallelism,

thus increasing the amount of vulnerable state. AVF throttling is based on the key observation that while exploiting concurrency on the critical path can significantly improve performance, the majority of the program has abundant *slack* and can be deferred to substantially reduce the amount of vulnerable state with negligible effect on the execution time. Our evaluation in the TRIPS architecture shows that around 90% of the vulnerable state is due to slack. We design a hybrid AVF throttling technique that uses the compiler to estimate slack and the hardware to dynamically exploit it. Using the compiler for static slack estimation considerably reduces the complexity of the technique. Further, it takes advantage of the TRIPS execution model and on-chip networks to exploit slack more efficiently, and significantly improves reliability by 25-42% for a set of SPEC and EEMBC benchmarks. We also present a detailed comparison of AVF throttling with prior approaches including redundant execution, and selective redundant execution. Based on the comparison, we argue that while AVF throttling may provide a smaller absolute reliability improvement, it significantly reduces the power consumption and complexity overhead, making the three techniques appropriate in systems with different reliability requirements.

Overall, this dissertation establishes that distributed architectures provide a good foundation for building a reliable system from unreliable components, and our results set a good starting point for further innovative research in this area.

Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xvi**

List of Figures **xviii**

Chapter 1 Introduction **1**

1.1 Approaches to Achieving Reliability 2

1.2 Shift toward Distributed Architectures 6

1.2.1 The TRIPS Architecture 8

1.3 Thesis Statement 9

1.4 Dissertation Contributions 9

1.5 Dissertation Organization 12

Chapter 2 Impact of Technology Trends on Hard Error Rate **14**

2.1 Background 16

2.1.1 Sources of Yield Loss 16

2.1.2 Design for Yield 17

2.2 Yield Modeling 18

2.2.1 Random Defect Limited Yield Model 19

2.2.2 Chip Area Model 20
2.2.3 Overall Chip Yield Model 22
2.3 Yield Projection 24
2.3.1 Chip Topologies 24
2.3.2 Results 26
2.4 Summary 28

Chapter 3 Impact of Technology Trends on Soft Error Rate 30
3.1 Background 32
3.1.1 Particles that Cause Soft Errors 32
3.1.2 Soft Errors in Memory Circuits 34
3.2 Soft Errors in Combinational Logic 36
3.3 Methodology 39
3.3.1 Device Scaling Model 39
3.3.2 Charge to Voltage Pulse Model 41
3.3.3 Electrical Masking Model 42
3.3.4 Pulse Latching Model 49
3.3.5 Latching-window Masking Model 50
3.3.6 Estimating SER of Combinational Logic 52
3.4 Results 54
3.4.1 Circuit Soft Error Rate 54
3.4.2 Processor Soft Error Rate 60
3.5 Discussion 63
3.6 Related Work 67
3.7 Summary 71

Chapter 4 Reliability of Distributed Architectures 73
4.1 Distributed Architecture Principles 74

4.1.1 Modular Design . . . 75
4.1.2 Cooperative Hardware/Software Techniques . . . 76
4.1.3 On-chip Networks . . . 78
4.1.4 Summary . . . 78
4.2 The TRIPS Architecture . . . 79
4.2.1 Modular Design . . . 79
4.2.2 Cooperative Hardware/Software Techniques . . . 81
4.2.3 On-chip Networks . . . 85
4.2.4 Summary . . . 86
4.2.5 Events in the Execution of a TRIPS Block . . . 88
4.3 Synergy between Distributed Architecture Principles and Reliability . . . 90
4.3.1 Fault Avoidance . . . 92
4.3.2 Fault Tolerance . . . 93
4.3.3 Fault Evasion . . . 95
4.3.4 Performance-Reliability Tradeoff . . . 97
4.4 Summary . . . 98

Chapter 5 Techniques for Improving Hard Error Reliability 102

5.1 Performance-Averaged Yield . . . 104
5.2 Exploiting Microarchitectural Redundancy for Defect Tolerance in Dynamic Architectures . . . 106
5.2.1 On-chip Redundancy Model . . . 107
5.2.2 Extensions to the Overall Chip Yield Model . . . 111
5.2.3 Results . . . 112
5.3 Fault-Aware Instruction Scheduling Heuristics for Static Architectures . . . 122
5.3.1 Design Space for Fault Reconfiguration in Static Architectures . . . 122
5.3.2 Fault Reconfiguration in the TRIPS Architecture . . . 124
5.3.3 Fault Model . . . 127

5.3.4 Evaluation Methodology 129
5.3.5 TRIPS Block Utilization 131
5.3.6 Fault-Aware Instruction Placement 135
5.3.7 Compiler-Assisted Fault Reconfiguration for Improving Lifetime Reliability 141
5.3.8 Tradeoff between Block Utilization and Algorithm Complexity . . 141
5.3.9 Hybrid Hardware-Software Approach for Fault Reconfiguration . . 142
5.4 Related Work 143
5.5 Summary 146

Chapter 6 Soft Error Reliability Regimes 148

6.1 Soft Error Rate Scaling Analysis 151
6.2 Computing the Architectural Vulnerability Factor 155
6.3 Reliability Performance Ratio (RPR) 158
6.4 Soft Error Reliability Regimes 160
6.4.1 Error Correcting Codes 160
6.4.2 Full Redundant Execution (FRE) 161
6.4.3 AVF Throttling 163
6.4.4 Selective Redundant Execution (SRE) 165
6.4.5 Summary 166
6.5 TRIPS AVF Methodology and Evaluation 167
6.5.1 TRIPS AVF Methodology 168
6.5.2 TRIPS Baseline AVF Results 174
6.6 Discussion 177
6.7 Summary 178

Chapter 7 Techniques for Improving Soft Error Reliability 180

7.1 Full Redundant Execution (FRE) 181

7.1.1 Evaluation of Redundant Execution 186
7.1.2 Accelerating Dataflow Execution in the Redundant Block 188
7.1.3 Predication Model Optimization 192
7.1.4 Evaluation of Redundant Execution Optimizations 193
7.2 AVF Throttling 195
7.2.1 Dynamic Speculation Control (DSC) 201
7.2.2 Fetch-on-Demand (F-o-D) 206
7.2.3 Comparison of Dynamic Speculation Control and Fetch-on-Demand 213
7.2.4 Measuring Actual Reduction in Instruction Slack ACE Cycles . . . 214
7.2.5 Using Reliability Performance Ratio to Tune AVF Throttling 218
7.3 Selective Redundant Execution (SRE) 220
7.4 Summary 224

Chapter 8 Conclusions **231**
8.1 Hard Error Reliability 232
8.2 Soft Error Reliability 233
8.3 Final Thoughts 235

Bibliography **238**

Vita **260**

List of Tables

2.1 Uniprocessor Area Model: Percentage contributions of the different microarchitectural structures to the total area. 22

2.2 Multiprocessor configurations: Increasing number of processor cores with decreasing feature size. Since each core has a private L2 cache, it occupies a relatively constant fraction of the total chip area across technologies . . . 26

3.1 Key characteristics of CMOS device models 41

3.2 Switching voltage of the gates for the rise and the falling transitions at different process technologies 45

3.3 Critical charge for an SRAM cell/latch/logic chain by feature size and pipeline depth 58

3.4 Transistors per chip for 16 FO4 pipeline using quadratic scaling assumption 61

3.5 Chip model for 350nm device size 61

4.1 TRIPS processor micronetworks 86

4.2 Composition of the processor tiles. 91

4.3 Features of the TRIPS architecture that aid in implementing the principles of fault tolerance for hard error reliability 100

4.4 Features of the TRIPS architecture that aid in implementing soft error reliability 101

5.1 Processor redundancy configuration 110
5.2 Benchmarks used for performance experiments 113
5.3 Relative IPCs for a few sample degraded configurations 115
5.4 Determining the fault model using the yield distribution for the ETs 128
5.5 List of EEMBC benchmarks used for evaluation. 130
5.6 List of SPEC benchmarks used for evaluation. 131
5.7 Dynamic block capacity of a set of EEMBC, SPEC integer and floating point benchmarks. 133

6.1 TRIPS processor structures for which ACE cycles are tracked 168
6.2 Bit fields of a read queue entry 170
6.3 Bit fields of a write queue entry 171
6.4 Bit fields of an OPN control packet 172
6.5 Bit fields of an OPN data packet 173
6.6 Baseline AVF results 175

7.1 Using RPR to tune dynamic speculation control 219
7.2 Comparison of the reliability improvement and design tradeoffs between the different regimes 226
7.3 Synergy between AVF and EPI throttling techniques 228

List of Figures

2.1 Yield loss components. 17

2.2 Yield VS time curve of a typical product. 18

2.3 Uniprocessor Model: Baseline model of the uniprocessor. Processor core similar to the Alpha 21264. 21

2.4 Chip topologies: a) Constant-architecture model has decreasing chip area with reducing feature size, b) Constant-area model uses a chip multiprocessor design methodology with greater number of cores at smaller feature sizes occupying relatively constant area 25

2.5 Impact of technology scaling and processor model on chip yield 27

2.6 Defect densities required to achieve 83% ITRS target for random-defect limited yield 28

3.1 Particle flux: Energy distribution of neutron flux from cosmic rays, with energy level on the x-axis and the flux on the y-axis. The most important aspect is that, particles of lower energy occur far more frequently than particles of higher energy. 33

3.2 Simple model of a pipeline stage used as the primitive circuit for estimating SER of combinational logic 37

3.3 Circuit diagram of a pipeline latch 38

3.4 Process for determining the Soft Error Rate in a logic chain 40

3.5 A current pulse resulting from a particle strike at a device node 43
3.6 The delay degradation effect 46
3.7 Model to compute rise/fall time based on gate delay 48
3.8 Latching-window masking 51
3.9 Computing SER using a range of charges with varying probability of latching. 54
3.10 SER of individual circuits 55
3.11 Critical charge for a single SRAM cell/latch/logic chain 57
3.12 Ratio of critical charge to charge collection efficiency for SRAM/latch/logic 59
3.13 Effect of latching-window masking 60
3.14 SER/chip for SRAM/latches/logic 63
3.15 Circuit implementations: a) Static 3-input NAND gate, b) CMOS domino logic : latched version 65

4.1 TRIPS processor microarchitecture 80
4.2 Instruction physical layout 84
4.3 TRIPS micronetworks 87
4.4 Events in the execution of a TRIPS block 88

5.1 Basic redundancy models 107
5.2 IPC distribution for the different configurations 114
5.3 Yield for a constant-architecture uniprocessor model at normal defect size . 116
5.4 Yield with intra-processor redundancy at normal defect size 118
5.5 Yield with inter-processor redundancy at normal defect size 119
5.6 Comparison of Y-PAV for different redundancy models 120
5.7 Fault-aware instruction placement 129
5.8 Performance sensitivity to block capacity for the set of EEMBC benchmarks 134

5.9 Fault reconfiguration by rescheduling a pre-placed instruction, to create a new location for the current instruction which provides it with functional routes from both its parents 137
5.10 Rescheduling a pre-placed parent instruction changes the configuration such that it allow the current child instruction to be successfully scheduled 138
5.11 Defective execution tiles at the register tile and data tile boundary can make it impossible to access a particular register or data cache bank. Dedicated bypass logic, software support, or programmable hardware support is required to handle these defects. 139

6.1 Mukherjee et al. proposed this classification of a soft error in a bit based on its severity [2]. SDC = silent data corruption, DUE = detected unrecoverable error. 149
6.2 Soft error rate scaling analysis 154
6.3 TRIPS instruction formats 169

7.1 Block-granular redundant execution model 182
7.2 Using the LSQ to perform replication of load values and comparison of stores, and the BOQ to compare branch addresses 183
7.3 Impact on execution cycles and ACE cycles if the maximum number of speculative blocks is reduced from eight to four 187
7.4 Impact on execution and ACE cycles for SPEC integer benchmarks with full redundant execution 189
7.5 Impact on execution and ACE cycles for SPEC floating point benchmarks with full redundant execution 189
7.6 Impact on execution and ACE cycles for EEMBC benchmarks with full redundant execution 190

7.7 Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with full redundant execution 190
7.8 Decoupling fault detection and load value delivery in the CRB 191
7.9 Using predication at the bottom in primary blocks for higher performance, and predication at the top in the CRB for redundantly executing fewer instructions. 193
7.10 Effect of optimizations applied to redundant execution 195
7.11 ACE cycles characterization 199
7.12 Impact of speculation depth on execution and ACE cycles 202
7.13 Impact on execution and ACE cycles for SPEC integer benchmarks with dynamic speculation control 204
7.14 Impact on execution and ACE cycles for SPEC floating point benchmarks with dynamic speculation control 204
7.15 Impact on execution and ACE cycles for EEMBC benchmarks with dynamic speculation control 205
7.16 Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with dynamic speculation control 205
7.17 Exploration of potential performance-reliability tradeoff with fetch-on-demand. 208
7.18 Impact on execution and ACE cycles for SPEC integer benchmarks with fetch-on-demand 211
7.19 Impact on execution and ACE cycles for SPEC floating point benchmarks with fetch-on-demand 211
7.20 Impact on execution and ACE cycles for EEMBC benchmarks with fetch-on-demand 212
7.21 Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with fetch-on-demand 212
7.22 Comparison of AVF throttling mechanisms 215

7.23 Impact on execution and ACE cycles for SPEC benchmarks with the AVF throttling techniques combined 215
7.24 Impact on execution and ACE cycles for EEMBC benchmarks with the AVF throttling techniques combined 216
7.25 Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with the AVF throttling techniques combined 216
7.26 Actual reduction in instruction slack ACE cycles 217
7.27 Dynamic speculation control based selective redundant execution 221
7.28 Impact on execution and ACE cycles for SPEC integer benchmarks with selective redundant execution 222
7.29 Impact on execution and ACE cycles for SPEC floating point benchmarks with selective redundant execution 222
7.30 Impact on execution and ACE cycles for EEMBC benchmarks with selective redundant execution 223
7.31 Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with selective redundant execution 223
7.32 Comparison of the three soft error reliability regimes: full redundant execution (FRE), selective redundant execution (SRE), and AVF throttling. Graph shows that the reliability improvement from AVF throttling is comparable to selective redundant execution. 225

Chapter 1

Introduction

The bulk of the performance improvement in microprocessors has come from aggressive scaling of devices coupled with innovations in architecture. Smaller feature sizes lead to faster transistors and permit a greater number of transistors in a given chip area. During the last sixteen years, feature sizes have scaled from 1000nm to 65nm, and the amount of logic per pipeline stage has decreased from 84 to 12 FO4 contributing to a 60-fold increase in clock frequency and performance improvement in the Intel family of processors. Architects have taken advantage of the decreasing feature sizes to increase on-chip transistor count from about 1 million at 1000nm to approximately 300 million at 65nm. Successive processor designs have used the larger number of transistors to enable greater on-chip computation and storage capacity for higher performance. To maintain a competitive edge, semiconductor manufacturers have thus constantly used processes that are still immature and have undergone tremendous changes over time, and forecasts project continued geometry shrinks to at least 22nm within the next decade [3].

However, shrinking lithography, new materials and process technologies, stricter design tolerances, constant or increasing die sizes, and higher levels of on-chip integration make integrated circuits more susceptible to manufacturing defects [4]. Further, relentless scaling, and rising power and thermal densities are expected to significantly affect processor

lifetimes [5]. Finally, smaller feature sizes and higher levels of on-chip integration are also expected to increase the rate of soft errors, which are errors in processor execution due to external radiation or noise rather than design or manufacturing defects. Together these factors increase both the vulnerability of individual transistors, and the total number of transistors that can have errors. Based on these projections, the International Roadmap for Semiconductors has outlined processor hard and soft error reliability as key emerging technology challenges that must be solved effectively at multiple levels of the design [4].

1.1 Approaches to Achieving Reliability

Achieving system reliability requires effort at all levels of the design. While techniques at the device and circuit level have finer control over the susceptibility of individual circuits to errors, architectural techniques may have lower overhead since they are amortized over a larger amount of state, and hence can also exploit the fact that not all errors affect the final program outcome. Further, reliability cannot be based only on one-time factory testing at future technologies, and dedicated mechanisms that operate during the lifetime of the chip will be required [6]. Heimerdinger et al. proposed a conceptual framework for system reliability [7], and divided the approaches for improving reliability into four categories: fault avoidance, fault removal, fault tolerance, and fault evasion. The next few paragraphs explain each of these categories. We discuss techniques from each of the categories, some that are already used in modern designs, and some that have been proposed in research.

Fault Avoidance and Removal: Fault avoidance refers to techniques that are used at design time to reduce the system's baseline susceptibility to faults. Design rule induced systematic yield losses can be avoided by filling otherwise unused tracks in the layout with metal to establish a regular pattern and enable better equipment calibration during manufacturing [6]. Most modern chip designs use silicon-on-insulator (SOI) technology that provides many advantages including significantly reduced susceptibility to soft errors due

to the smaller volume for charge collection [8, 9]. Judicious gate sizing and capacitance insertion at the layout and CAD level can improve the tolerance to soft errors with negligible cost in performance [10, 11]. Burn-in is a fault removal technique that weeds out defective chips after manufacturing time, so that chips actually used in systems have a very low failure rate [12, 13]. Chips also typically include *design for test* (DFT) structures such as scan-chains to aid in fault removal.

Fault Tolerance: Fault tolerance refers to techniques that maintain the the availability of the system, perhaps at a degraded level, in the presence of a fault. White et al. defined four necessary and sufficient conditions that a technique must meet in order to provide a system with fault tolerance [14]:

- **Redundancy:** In general, there are three types of redundancy - spatial, information, and temporal redundancy. In spatial redundancy, duplicate units are used to either verify original execution, or replace defective units to improve system reliability. Information redundancy uses dedicated hardware to generate extra code bits that can be used to check the integrity of the data. Depending on the sophistication of the code, information redundancy can be used both for error detection and correction. Unlike spatial and information redundancy, temporal redundancy verifies execution by performing it multiple times on the same hardware, and hence cannot tolerate a permanent defect in the unit.

  Therefore, only spatial and information redundancy are relevant in the context of hard errors in which a particular device becomes permanently defective. For instance, redundant rows and columns are typically included with memory arrays to take the place of defective ones, with modest extra logic in the address decoder to perform this reconfiguration. Similarly, information redundancy in the form of ECC and parity are commonly used to handle bit defects within a single row [15]. Today's systems typically provide fail-in-place capabilities at the system level by including hot spares

for power supplies, processor chips, memory modules, and disks [16].

Spatial, temporal, and information redundancy are all applicable to soft errors since they are transient in nature, and don't have any permanent effect on the device function. Several architectural techniques for redundant execution have been proposed in research taking advantage of these three types of redundancy [17–19]. Redundancy can also be applied at the circuit or device level in the form of radiation-hardening, but can incur up to 100% overhead [20]. On the other hand, architectural techniques such as ECC and redundant execution typically have fewer overheads of 13% and 30% respectively [19, 21].

- **Fault detection and annunciation:** A key component of fault tolerance is to detect the fault and communicate the defective configuration to the rest of the system. External testers or built-in-self-test (BIST) hardware can be used to detect and communicate information regarding hard errors to hardware or firmware to perform fault reconfiguration [22–24]. Structures which use ECC or parity perform error detection by checking the integrity of the data bits using the code bits on every access [15]. The advantage of error correcting codes is that they can detect both hard and soft errors. Finally, redundant execution has been used for fault detection in systems that require very high reliability [18, 25].

- **Fault isolation:** Fault isolation ensures that a detected fault is contained and not allowed to propagate to the rest of the system. For hard errors, this is typically implemented using BIST controllers which detect errors in a specific module. For soft errors, mainstream server processors use many techniques including but not limited to *machine check abort* on double bit errors in SEC-DED ECC protected state, or single bit errors in parity protected state before the bad data is consumed, and data poisoning of cache lines that store data with double-bit ECC errors from the system bus [15]. Data poisoning helps to only terminate processes that consume the erroneous data, while keeping the rest of the system functioning. Aggarwal et al.

propose important changes that can be made to commodity multi-core architectures, with modest additions in hardware, to significantly improve fault isolation and increase their availability [26].

- **On-line repair:** Fault recovery or repair is the final step in achieving fault tolerance once the fault has been detected and isolated. There are many techniques used in systems today for fault recovery. Commonly used SEC-DED ECC is capable of both detecting and correcting single bit hard and soft errors. As described earlier, the reconfiguration logic in address decoders enable the use of redundant rows and columns in memory arrays. Many high end systems also support disabling an entire memory module or cache if they are affected by unrecoverable errors in the module [15, 27]. Further, high end IBM servers also use a technique called dynamic processor sparing to transparently replace a defective processor with a fully functional spare processor [28]. Such fault recovery techniques typically also require periodic checkpointing of processor state, so that execution can be restarted from a known, error-free, checkpoint when a hard or soft error is detected [24, 28–30].

Fault Evasion: Fault evasion is similar to fault avoidance, but is applied during the processor lifetime. Fault evasion techniques observe the dynamic behavior of the system for periods of high vulnerability to hard or soft errors, and trigger preventive reconfiguration to reduce the probability of occurrence of an error. Dedicated hardware or software monitors for symptoms that are either direct indicators of an error that has already occurred, or indirect indicators that signal periods of high vulnerability to errors. In the context of hard errors, fault evasion may involve preventively isolating the failure-prone resource; and for soft errors the system may proactively transition to a more reliable execution mode. IBM multiprocessor systems monitor processor failures using firmware routines, and trigger reconfiguration if the number of errors exceed a particular threshold [28]. Srinivasan et al. proposed dynamic reliability management (DRM) as a fault evasion technique to improve

the lifetime reliability of a processor [31]. Wang et al. proposed symptom based soft error detection using symptoms such as exceptions, and mispredictions on high-confidence branches to trigger checkpoint-based fault recovery [29]. Fault evasion can also be applied at the circuit level by monitoring circuit wear out [32].

1.2 Shift toward Distributed Architectures

Several technology trends have become first order processor design constraints, and have impelled a shift toward distributed architectures with smaller peak frequencies and pipeline depths.

Global wire delays: The delay of long wires used to access large storage structures, or that communicate across multiple modules, or pipeline stages, and whose lengths do not correspondingly scale with feature size, will increase relative to gate delays with each successive technology generation [33, 34]. Therefore, the clock frequency of an architecture that uses such global communication will be limited by these wire delays at future technologies.

Power consumption: Without explicit power management techniques, aggressive technology scaling, higher frequencies, and greater on-chip integration will lead to increasing dynamic and static power consumption every generation. Dynamic power consumption will increase both because of the greater number of transistors switching every cycle, and the larger capacitive load per transistor. Similar to dynamic power, static power consumption increases with greater on-chip integration, but is also exponentially dependent on the operating temperature. Since operating temperature is itself a function of the total power consumption, static power becomes part of this dangerous feedback loop.

Limits on deep pipelining: Hrishikesh et al. established that reducing the logic depth of a pipeline stage below 6-8 FO4 gate delays, and thus increasing the overall pipeline

depth, leads to negative returns in performance. They demonstrated that this effect is due to both increased latency of the critical loops (branch mispredict loop, load-to-use loop, and issue-wakeup loop), and the increased overheads of the latching window, clock skew, and jitter that limit the useful work per clock cycle [35]. Architectures with deep pipelines also traditionally employ complex all-to-all bypass networks between interacting pipeline stages, and speculative control and data flow techniques to manage the increased latency of the critical loops [36]. Hence, they exacerbate the issues of global wire delays, and high power consumption.

Based on these technology trends, we identify three main principles on which distributed architectures must be built to achieve technology scalability with respect to performance, power consumption, and design complexity. The three main principles are the following which are developed in greater detail in Chapter 4:

Modular design: Employ resource partitioning at different granularities to eliminate access to large centralized resources, and limit wire lengths and pipeline depths within each module. Microarchitecture is composed by connecting the different modules as necessary, and high performance is achieved by exploiting concurrent execution across the modules. The emphasis on concurrency reduces the pressure on clock frequency and can decrease power consumption.

On-chip interconnection networks: Utilize well-defined, scalable networks for communication between the different modules. These networks will likely be point-to-point or have restricted connectivity, and multi-hop with each hop taking a clock cycle. They must also implement explicit communication protocols for managing the global latency between modules, and techniques for concurrency such as pipelining to improve network throughput.

Cooperative hardware / software techniques: Distributed architectures must also rely on greater cooperation between the hardware and the software to mine parallelism from the application and convey it efficiently to the distributed hardware, which can then concurrently execute the parallel components. Using a hybrid approach instead of pure hardware techniques to extract parallelism can potentially improve the power consumption and performance scalability of distributed architectures.

We observe that there is synergy between the principles of distributed architectures and the conditions that a system must meet for being reliable. Modular design, on-chip networks, and hybrid execution models are useful building blocks for improving hard and soft error reliability. Further, these features can also be used to configure the tradeoff between reliability and the overhead in performance and power consumption to suit a range of reliability requirements.

1.2.1 The TRIPS Architecture

The TRIPS architecture is a technology scalable distributed architecture that we use in this dissertation for evaluating our mechanisms. While Chapter 4 describes the architecture in detail, some salient features of the architecture are presented here:

Tile-based design: The TRIPS architecture uses a tile-based modular design. All major processors structures including but not limited to the register file, the instruction window, and on-chip caches are partitioned into multiple *tiles*. Each type of tile is used multiple times as necessary, and connected together to compose the overall microarchitecture.

On-chip networks: The tiles are inter-connected using point-to-point, multi-hop, nearest-neighbor networks for improved scalability of communication.

Block-atomic execution model: Unlike conventional architectures that use an instruction-atomic execution model, TRIPS treats a block of instructions together as a single atomic unit

in the pipeline to amortize the per-instruction bookkeeping and logic overheads. Within a block the TRIPS ISA provides explicit support for direct communication between dependent instructions, eliminating the need for conventional associative look-up hardware for instruction wake-up that does not scale to large windows at future technologies.

1.3 Thesis Statement

This dissertation proposes architectural techniques to improve the hard and soft error reliability of modern and future technology-scalable distributed architectures. This dissertation demonstrates with the help of detailed end-to-end models that both chip hard and soft error rates increase substantially at future technologies, and emphasizes the need for innovative solutions to maintain chip reliability at acceptable levels with low overhead. It identifies key synergies and extra demands placed by important new features of distributed architectures including on-chip networks and the greater reliance on software, that must be considered to achieve high reliability at low overhead. Specifically, the dissertation presents a detailed evaluation and comparison of the reliability improvement, performance overhead, and other design tradeoffs of hard and soft error reliability mechanisms in a distributed architecture.

1.4 Dissertation Contributions

This dissertation makes the following main contributions.

Impact of Technology Trends on Hard and Soft error rate: Based on detailed models for defect distribution, yield, chip area, and available redundancy in different chip components we project the random defect limited yield at future technologies in the context of both a uniprocessor and a chip multiprocessor. We show that with only redundant rows and columns in memory arrays yield drops significantly from 85% at 250nm to 60% at 50nm. We describe and validate an end-to-end model that enables us to compute the soft error rates

(SER) for existing and future microprocessor-style designs. The model captures the effects of important masking phenomena, electrical masking and latching-window masking, which inhibit soft errors in combinational logic. We quantify the SER due to high-energy neutrons in SRAM cells, latches, and logic circuits for feature sizes from 600nm to 50nm and clock periods from 16 to 6 fan-out-of-4 inverter delays. Our model predicts that the raw SER per chip of combinational logic circuits will increase three orders of magnitude from 180nm to 50nm, and at that point will be comparable to the SER per chip of latches, and will be within two orders of magnitude of unprotected memory elements.

Synergy between Principles of Distributed Architectures and Reliability: We identify three key underlying principles of distributed architectures that enable technology scalability: a) modular design, b) on-chip interconnection networks, and c) cooperative hardware/software techniques. Modular design and on-chip networks provide the foundation for abundant spatial redundancy for computation and communication, that can be used to improve reliability. Further, the greater reliance on software to expose concurrency in the application, and the microarchitectural protocols for exploiting the concurrency in the distributed architecture together provide a natural foundation for managing the available redundancy efficiently to improve reliability at low overhead. We present a detailed qualitative analysis of the synergies between these three principles and the design requirements for fault avoidance, fault tolerance, and fault evasion. Specifically, we identify the salient features of the TRIPS architecture that can be exploited to improve reliability at low overhead, tile-based modular design, multiple dedicated point-to-point on-chip networks for different processor functions, block-atomic, and static-placement-dynamic-issue (SPDI) execution model, and direct instruction communication.

Yield Enhancement Techniques for Dynamic and Static Architectures: We propose to use the inherent redundancy available in existing and future distributed architectures to improve yield and enable graceful performance degradation in fail-in-place systems. We

advocate pushing fail-in-place inside the boundaries of a single chip or processor, and show that mechanisms already exist in dynamic architectures that can easily disable the defective components from being used during program execution. We introduce a new yield metric called *performance averaged yield (Y_{PAV})* which accounts both for fully functional chips, and those that exhibit some performance degradation due to defective components that have been disabled. By exploiting microarchitectural redundancy we demonstrate that Y_{PAV} can be improved to as high as 99.6% at 50nm, with a maximum reduction in performance in any chip of less than 20% on a suite of SPEC benchmarks, a substantial improvement from a yield of 60% achieved when only considering the defect-free parts.

While defects in dynamically scheduled architectures can be tolerated using mechanisms that are transparent to software, static architectures create different opportunities and challenges for reliability management. This dissertation proposes to expose the defective hardware configuration in a static architecture to the compiler, which can perform efficient fault reconfiguration through intelligent fault-aware instruction scheduling. We conducted our studies on the TRIPS architecture, and demonstrate that for two specific fault models, the fault-aware scheduling heuristics we propose exploit the redundancy in the TRIPS hardware to successfully reschedule the full set of SPEC and EEMBC benchmarks, with less than 4% impact on performance. While we primarily focus on hard error reliability in the context of static chip yield, we also discuss how these techniques can be applied to improve processor lifetime reliability [31, 37].

Soft Error Reliability Regimes: The soft error reliability requirement and the acceptable overhead of reliability mechanisms vary depending on the application and market segment. Based on this understanding, we provide a systematic methodology for quantifying the soft error reliability improvement needed every technology generation, and discuss the performance-reliability tradeoffs of achieving it at different levels of the design including device techniques, error correcting codes, and architectural mechanisms. We propose Reliability Performance Ratio (RPR), a new metric which a designer can use to measure the

performance-reliability tradeoff, and achieve the reliability target with the least excessive protection and overhead.

Traditionally, systems have achieved high soft error reliability using sophisticated error correcting codes (ECC), or redundant execution on duplicate processors or pipeline stages, but have also incurred correspondingly high cost. Building on this new framework, we present two better-than-worst-case soft error reliability regimes, one called AVF Throttling, a class of microarchitectural fault evasion techniques that efficiently reduce the amount of vulnerable processor state, and Selective Redundant Execution which relies on redundant execution of specific sub-portions of the program to provide high reliability at low overhead. We demonstrate that both AVF throttling, and selective redundant execution achieve reasonable reliability improvement, but with significantly better performance-reliability tradeoff than redundant execution. We also demonstrate the effectiveness of RPR by configuring the AVF throttling techniques to two different operating points; to a system for which performance and reliability are equally important, and a second performance-centric system.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents the yield modeling framework and projections at future technologies. Chapter 3 builds an end-to-end model for estimating the soft error rate of combinational logic, and presents the soft error rate projections for SRAM cells, latches, and combinational logic for different technologies and pipeline depths. Chapter 4 describes the TRIPS architecture in detail, and presents a detailed analysis of the synergies between the principles of distributed architectures and reliability. In the rest of the chapters, we explore a subset of these synergies to build and evaluate mechanisms for improving hard and soft error reliability.

Chapter 5 describes architectural techniques for improving yield that capitalize on the inherent redundancy of distributed architectures, and some specific features of the

TRIPS architecture. It also qualitatively discusses how these mechanisms can be adapted to improve processor lifetime reliability. Chapter 6 builds a systematic framework for computing the chip-level soft error rate accounting for microarchitectural and architectural masking factors, and extends this analysis to quantify the performance-reliability tradeoff of a reliability technique. Building on this framework, it presents four soft error reliability regimes with different performance-reliability tradeoffs. Chapter 7 evaluates specific techniques from each of these regimes in the TRIPS architecture, and presents a comparison of their performance-reliability tradeoff and other overheads. Finally, Chapter 8 concludes with some broad principles for building reliable systems and points to some interesting future directions.

Chapter 2

Impact of Technology Trends on Hard Error Rate

While technology trends suggest chips with clock frequencies in the multi-gigahertz range containing over a billion transistors by the end of the decade, two substantial challenges must be addressed to enable practical deployment of such systems. First, shrinking lithography, new materials and process technologies, lower design tolerances, constant or increasing die sizes, and higher levels of on-chip integration make integrated circuits more susceptible to manufacturing defects, requiring substantial investments to maintain chip yield at acceptable levels. The Semiconductor Industry Association has set a target of 75% (75 good chips for every 100 manufactured) for overall microprocessor yield [38]. While manufacturing engineers have made substantial innovations in materials and clean rooms to achieve this target at current technologies, realizing this target in future processes may prove extremely costly. Modern fabrication facilities cost more than $10 billion to build and equip, in part because of the need to reduce and eliminate contamination in the factory caused by workers, equipment, materials, and the air supply. Second, some manufacturing defects are latent and manifest themselves only after the chips have been deployed and run for some period of time. As larger commercial and scientific systems are constructed

from hundreds or thousands of processors, the probability and frequency of latent failures increase. Thus, hard errors can be divided into two main categories:

Extrinsic failures are caused by process or manufacturing defects, and have no direct relation to the processor microarchitecture. For example, contaminants on silicon or metal layers can lead to defective device operation, open, or short circuits. A technique called *burn-in* is used to test the manufactured devices at elevated operating temperatures and voltages to accelerate the occurrence of extrinsic failures [12, 13]. Traditionally, chips that pass burn-in will have very low extrinsic failure rate, but researchers have recently questioned whether a one-time testing methodology using burn-in will be viable at future technologies [6, 39].

Intrinsic failures are caused by processor wear out due to repeated use of the device [31]. For example, a wire narrowed by a manufacturing defect may function correctly during manufacturing and burn-in testing, but may degrade and break under repeated use. While the intrinsic failure rate is dependent on the process parameters and device materials, it is also very dependant on the operating temperature which is a function of the processor microarchitecture. Once an intrinsic failure occurs, succeeding errors will likely occur at the same location probably with increasing frequency unless corrective action is taken. Electromigration, and time dependent electric breakdown are examples of intrinsic failures [31].

Chip yield is becoming a significant problem with the increasing demand for on-chip resources that maintains die sizes relatively constant despite aggressive reductions in feature size. In this Chapter, we study the impact of technology scaling on chip yield in the context of both a uniprocessor and a chip multiprocessor (CMP) design. We acknowledge that intrinsic failures are also a growing concern [5], and recognize that many of the techniques developed in later chapters for reducing yield loss due to extrinsic failures can be adapted and applied at runtime to tolerate intrinsic failures and improve lifetime reliability [31].

The remainder of this Chapter is organized as follows. Section 2.1 provides background on the different kinds of yield loss. Section 2.2 describes the details of our yield, and area models, and Section 2.3 presents the projected yield at future technologies for different defect characteristics. Section 2.4 summarizes our findings and discusses possible ways to improve yield in future distributed architectures.

2.1 Background

Understanding yield loss is a critical activity in semiconductor device manufacturing. The overall yield is influenced by many factors, including the maturity of the fabrication process, the ability of a particular design to tolerate defects, and the ability to identify usable parts from unusable ones.

2.1.1 Sources of Yield Loss

Yield loss can be *functional*, that occurs when a device fails to meet the intended functionality from a logical point of view. Even a single functional defect in the critical circuitry of a chip can cause the entire chip to be considered bad. Alternatively, yield loss can be *parametric*, which occurs when otherwise functional devices fail to fall within the allowed range of acceptable electrical characteristics. Although the specifications typically allow for some margin in the acceptable electrical characteristics to account for normal process variation, any devices that fall outside of this proven range are considered unusable. Both types of yield loss can be caused by either *systematic* defects or *random* defects. Systematic defects result from problems in the manufacturing process such as contamination of materials or imprecise calibration of the equipment. Random defects, on the other hand, are the result of inevitable particle impurities in the air and are much more difficult to overcome. Figure 2.1 classifies yield loss into its different components.

Yield loss over time can be divided into an initial phase of technology deployment dominated by systematic failures, with an eventual crossover to a more mature phase

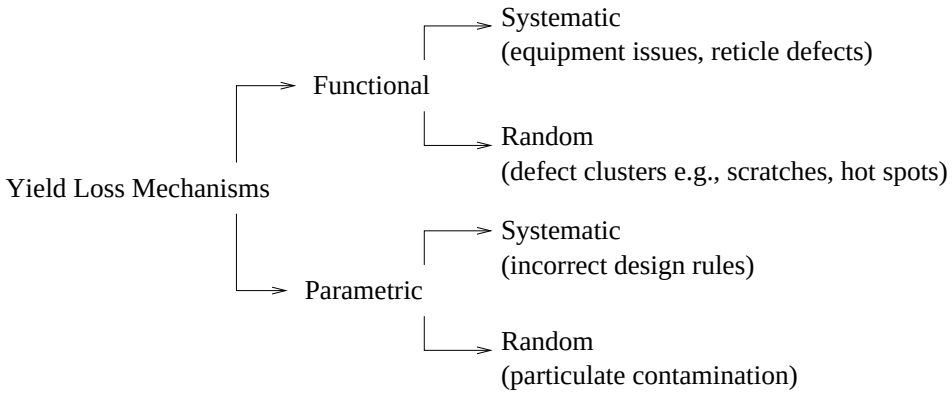

Figure 2.1: Yield loss components.

dominated by random defects [38]. Systematic defects can typically be overcome as yield learning occurs and the technology matures. Modern fabrication facilities invest enormous amounts of money into clean rooms to minimize particle density and size, but the ultimate yield is still limited by these random defects. Figure 2.2 illustrates the two phases of yield loss. Future technology advances are expected to involve continued shrinking of feature sizes, and the introduction of new process steps and materials increasing the yield sensitivity to design features, introducing new sources of systematic defects, and requiring a feature-based methodology to quantify yield loss [40]. As a result, manufacturers must either invest in more aggressive process control mechanisms and decrease the defect sensitivity of the designs, or accept elongated yield learning curves and lower final yields at future technologies. In general, as technology advances, the complexity and expense of yield management increases dramatically.

2.1.2 Design for Yield

The ground rules define the basic allowable structures in a particular technology. There are several ways in which they can be modified to incorporate additional *design for yield* guidelines that minimize the effects of common yield detractors [41]. For example, at the

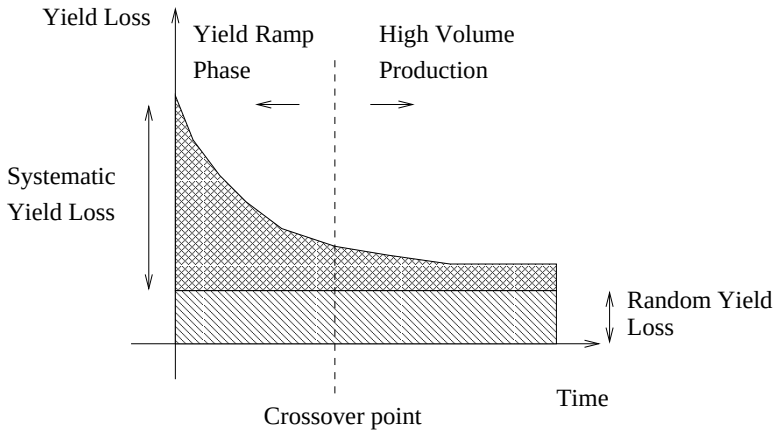

Figure 2.2: Yield VS time curve of a typical product.

circuit level, guidelines that discourage the use of tall stacked transistor structures and noise sensitive structures can help reduce parametric yield losses. In layout, filling otherwise unused tracks with metal to establish a regular pattern can enable better equipment calibration during manufacturing. Further, the use of redundant vias and fingered transistors form some degree of low-level defect tolerance in the design. For high volume manufacturing, feedback from yield analysis and iteration of the design to address these detractors is an important aspect of yield management. In designs with a high degree of regularity, such as DRAMs and SRAMs, it is common to make use of explicit redundancy to help improve yield. By including a small number of redundant rows and columns in the structure, along with steering logic in the decoders, the yield losses that could result from random defects in the main array can be minimized. A good overview of the different defect tolerance techniques used in VLSI circuits is provided in [41].

2.2 Yield Modeling

Our methodology for calculating overall chip yield integrates a basic yield model, and a microprocessor area model, with the redundancy model of the chip components. The *basic*

yield model is a probability distribution that calculates the random yield of a given area of silicon. The area of the chip components themselves are estimated using a *microprocessor area* model. The *yield with redundancy* model breaks a component into its redundant and non-redundant pieces before computing the component yield. The remainder of this section describes each of these models in greater detail.

2.2.1 Random Defect Limited Yield Model

Yield loss is a function of the size, material, location and the process step in which a defect is introduced. The Poisson and the Negative Binomial models are among the most commonly employed in the literature to relate the defect parameters to the resultant yield [41, 42]. The Poisson model assumes that the defects are completely independent. On the other hand, the Negative Binomial model uses a *clustering factor* to describe the degree of clustering of defects. It requires a specification of the microarchitectural granularity below which clustering of defects must be modeled, and above which defect occurrences can be assumed to be completely independent of one another.

In this study, we have adopted the Poisson Yield model for modeling the random yield component, because it allows us to simplify the mathematical treatment and focus more on the interaction between the redundancy models and the resulting yield trends. Since in reality the defects are not completely independent of one another, there is some error in the absolute values of yield that we predict in the study, though the magnitude of the error is not significant enough to undermine the usefulness of our results. The Poisson Yield (Y_P) Model models the random defects to be completely independent and is described by:

$$Y_P \propto e^{(-D0 \times A \times KR)} \tag{2.1}$$

where $D0$ is the defect density measured in defects per cm^2, A represents the area of the component in cm^2, and KR is the kill ratio or the fraction of the total component

area that is sensitive to defects. The equation below shows the dependence of the kill ratio on two factors:

$$KR \propto \left(\frac{Defect\ size}{Feature\ Size} \right) \tag{2.2}$$

The kill ratio models the interaction between the defect size and the layout feature size, and increases as the ratio of defect size to the feature size increases. The Poisson Yield model equation exhibits an exponential dependence of die yield on component area, defect density, and the kill ratio. Hence yield can be improved by either reducing the chip area or by reducing the defect density, and yield tends to get worse with smaller feature sizes because the kill ratio increases. The ITRS has set a target of 83% for the random-defect limited yield of microprocessors [38]. We obtain a Y_{BASE} of 85.4% at 250nm using the defect density provided by the ITRS, for a normal defect to feature size ratio, and a chip area of $320mm^2$, thus validating our input parameters to the Poisson Yield model.

2.2.2 Chip Area Model

The baseline uniprocessor architecture we use is modeled on the Alpha 21264 [43]. The only modification we make is to add an on-chip L2 cache with an associated shrink in the feature size. Figure 2.3 illustrates the uniprocessor model with the L2 cache coupled to the processor core. Estimation of individual component yield requires detailed area models of the processing cores and caches. To model the area of memory structures we used CACTI 3.0 [44], which accounts for the capacity, line size, associativity, number of ports and the technology generation, and returns the area and the area efficiency[1] of the structure. We configured CACTI 3.0 to derive area estimates of L1 and L2 caches, TLBs, register files, and all on-chip queues. To model the area of functional units we used an empirically derived, technology-independent area model [45]. We used a method of simple manual layouts to

[1] The area efficiency of a memory structure can be defined as the ratio of the area of the memory cells to the total area of the cache.

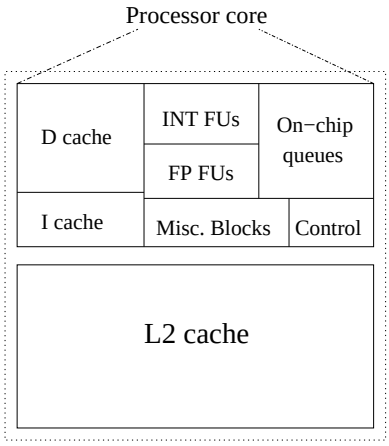

Figure 2.3: Uniprocessor Model: Baseline model of the uniprocessor. Processor core similar to the Alpha 21264.

estimate the area of random control logic components including the select, rename and instruction wakeup logic, based on the logic level block diagrams that are available in the literature [46]. To estimate the area of miscellaneous blocks such as I/O pads and clock distribution trees, we developed an empirical model based on our analysis of the Alpha 21264 floorplan [43].

The area model is designed to calculate the area of a processor hierarchically from the area of its individual components and uses the technology independent unit lambda for expressing the area. This design makes it easy to scale the model to different processor architectures and technology generations. We validated our area model against the Alpha 21264 microprocessor whose detailed floorplan statistics are available in [43]. The 16.7x18.8mm2 Alpha 21264 die was designed for a 350nm process and has 15.2 million transistors [47]. There is no on-chip L2 cache and both the L1 caches are 64KB. Alpha 21264 area from the floorplan is $314mm^2$, and the estimated area using the area model is $302mm^2$, an error of 3.8%. The source of this difference comes from unmodeled components of the chip that are not obvious from the die photos and floor plans. We believe that the areas of these components are so small that statistically their yield will be near perfect.

Structure	Percentage of total area
L2 cache	49.0%
L1 D cache	12.7%
L1 I cache	5.5%
Integer functional units	6.3%
Floating point functional units	6.7%
On-chip storage structures (except caches)	11.1%
Misc. components (BIU, PLL,I/O pads etc.)	6.0%
Random control logic	2.7%
Total Area at 250nm	$325mm^2$

Table 2.1: Uniprocessor Area Model: Percentage contributions of the different microarchitectural structures to the total area.

Using the area model, we calculated the area of the chip in Figure 2.3 to be $325mm^2$ at 250nm. Table 2.1 shows the area of the processor model, and its distribution among its most significant components.

2.2.3 Overall Chip Yield Model

To compute the yield of a component requires breaking it into its redundant and non-redundant pieces. For example, redundant rows and columns in memory structures provide coverage over defects that occur in the area occupied by the data cells, but a defect in the decode logic would still be fatal. This ability to distinguish between the regions included in the redundancy model and those that are still vulnerable to defects is fundamental to calculating the component yield.

In the current configuration, the chip shown in Figure 2.3 only supports Array Redundancy (AR) in the set associative L1 and L2 caches, and does not have any other protection against hard errors. When defects are detected in rows or columns of bit cells in the main body of the array, the AR mechanisms can be configured to effectively steer the decode towards the redundant entry rather than towards the bad row or column. This technique is already commonly used in many types of RAM chips as well as in the embedded RAM structures found in more general purpose chips such as microprocessors. From a

yield perspective, AR is attractive because a relatively small investment in area can offer excellent defect tolerance for the entire structure. In many cases, this can drive the yield loss due to these structures to very low levels with no loss in performance. Consistent with accepted design practice [48], the redundant rows and columns are about 2.5% of the base cache capacity.

Calculating Yield of caches with AR

Since defects in the Poisson Yield model are considered to be completely independent, the yield of the caches that have partial redundancy can be described by the equation:

$$Y = Y_{NR} \times Y_R \tag{2.3}$$

where Y_{NR} represents the yield for the area of the cache that has no redundancy, and Y_R is the yield of the cache array that is covered by AR. We use the efficiency metric for the SRAM array reported by CACTI 3.0 to evaluate the fraction of area devoted to the peripheral logic, the data, and tag arrays. The chip area model, and the poisson yield model can be directly used to compute the yield of the area without redundancy. The yield for the data and tag arrays, covered by the AR model are computed using a modified version of the Poisson model that accounts for the spare rows, and is described below.

The redundant rows in a cache are used only in the face of defects, and do not provide extra performance. The total number of rows in a cache is the sum of its base number of rows and the number of spare rows. In the Array Redundancy model, the cache is considered functional as long as it maintains its baseline capacity. Cache yield is then simply the probability that it has at least its baseline number of rows working out of the total number of rows including the spares. A particular cache configuration is specified by the number of functional rows, and can be achieved in multiple ways depending on exactly which of its rows are functional. The number of possibilities can be calculated using the *combinations* (C_r^n) operator. The overall cache yield is therefore the sum of

the probabilities associated with all the configurations in which it has at least the baseline number of functional rows, out of the total number of rows including spares. The Y_R from this calculation is summarized using the well known binomial expansion:

$$Y_R = \sum_{i\,=\,minr}^{(br+sr)} C_{minr}^{(br+sr)} \times \Pr(\text{Good})^i \times \Pr(\text{Bad})^{(br+sr-i)} \tag{2.4}$$

where C_r^n is the *combinations* operator, $minr$ is the minimum subset of rows required for correct functionality, br represents the base number of rows in the cache, and sr is the number of spare rows. The probability of a entry being functional or invalid is computed using the Poisson Yield model. For caches with AR, $minr$ is equal to br, and the value of sr is dependent on the cache capacity.

2.3 Yield Projection

This section presents our yield projection at future technologies and chip microarchitectures as a function of the defect characteristics. These results assume that the defect densities will remain constant at their value at 250nm with substantial investments in process control mechanisms. We begin by describing the two different chip topologies that we investigated, and then present the projected yield for both chip topologies.

2.3.1 Chip Topologies

Chip microarchitectures at future technologies have substantial flexibility in using the larger number of transistors that can fit in a given chip area. If the desired functionality and performance remain fairly constant with time, successive designs require few additional features in the processor architecture. In this study, we use the same uniprocessor architecture shown in Figure 2.3 across successive technology generations. As shown in Figure 2.4a, the area of the chip in the constant-architecture scheme decreases rapidly with decreasing feature size because the microarchitecture is kept constant.

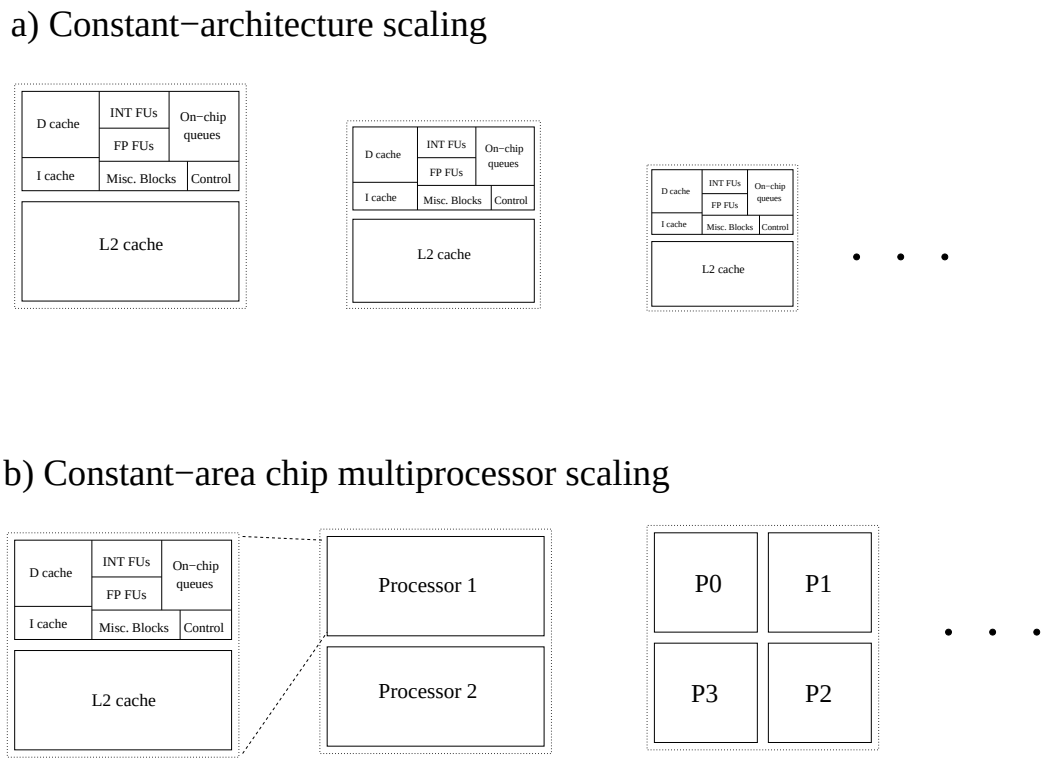

Figure 2.4: Chip topologies: a) Constant-architecture model has decreasing chip area with reducing feature size, b) Constant-area model uses a chip multiprocessor design methodology with greater number of cores at smaller feature sizes occupying relatively constant area

However, with successive microprocessor generations, the dominant trend in general purpose processor design has been to add microarchitectural features that enhance the processor's functionality and consume the extra silicon area. While this approach maintains a constant chip area, technology constraints and limits on available instruction level parallelism will lead to diminishing returns in performance if we naively scale the uniprocessor model across technologies. These technology scaling trends and considerations of multi-threaded performance have influenced many modern and emerging architectures to include multiple processors within a single chip to use the chip area more efficiently. While the design and configuration of each processor can change with time, in this study we use the uniprocessor shown in Figure 2.3 as the building block without loss of generality. Figure 2.4b illustrates the chip multiprocessor (CMP) model used in this study. The number of processors that can be accommodated per chip increases from 1 at 250nm to 24 at 50nm, given that the fraction of chip area consumed by L2 caches is kept approximately constant between 51-55% across all technologies. Table 2.2 lists all the CMP configurations we consider.

Technology	250nm	180nm	130nm	100nm	70nm	50nm
Processors	1	2	4	6	12	24
(L2cache Area)/(Chip Area)	51.23%	53.42%	52.47%	55.15%	52.60%	55.15%

Table 2.2: Multiprocessor configurations: Increasing number of processor cores with decreasing feature size. Since each core has a private L2 cache, it occupies a relatively constant fraction of the total chip area across technologies

2.3.2 Results

Since mainstream processors already employ redundant rows and columns in caches [16], we compute the *baseline yield* (Y_{BASE}) as the yield of a processor with AR in the L1 and L2 caches. Figure 2.5 plots Y_{BASE} of the chip multiprocessor (CMP) assuming that the defect characteristics are kept constant at their value at 250nm. The Y_{BASE} for the CMP with constant area decreases from 85% at 250nm to 60% at 50nm. Since the defect

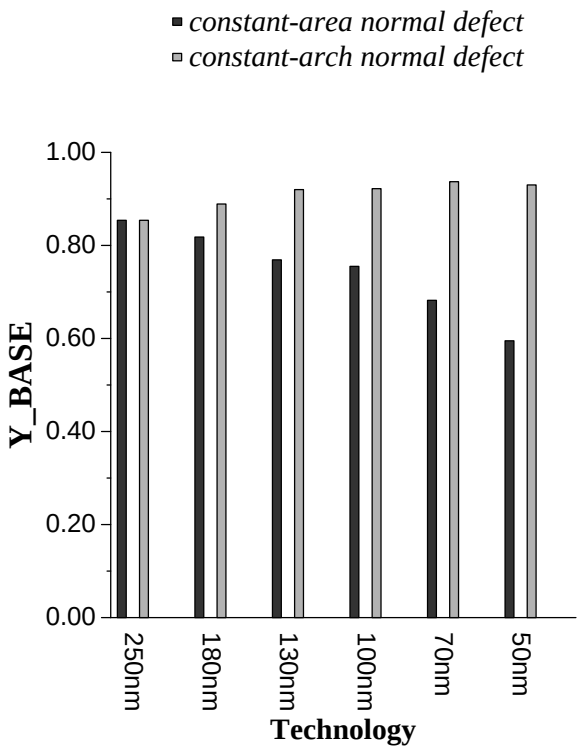

Figure 2.5: Impact of technology scaling and processor model on chip yield

characteristics are assumed to be constant, the kill ratio, which is inversely proportional to feature size, increases linearly with decreasing feature size and contributes to the substantial loss in yield. On the other hand, Y_{BASE} for a uniprocessor with constant architecture increases from 85% at 250nm to 93% at 50nm. This increase in yield is because the gain from the rapidly decreasing chip area outweighs the increased susceptibility to yield loss due to the higher kill ratio.

Figure 2.6 shows how defect densities must scale with technology to achieve the target 83% yield. While aggressive reductions in defect densities are required in the constant-area CMP model, larger defect densities can be tolerated at future technologies in the constant-architecture uniprocessor model. The reasons for this trend are logically the same as for Figure 2.5. Therefore, in the absence of microarchitectural redundancy, manufactur-

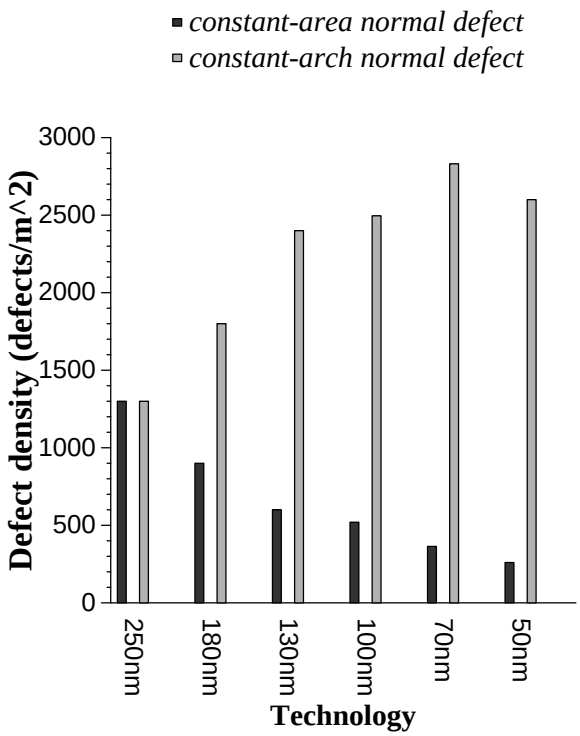

Figure 2.6: Defect densities required to achieve 83% ITRS target for random-defect limited yield

ers must either invest in more aggressive mechanisms to decrease the defect sensitivity in the designs or accept lower final yields at future technologies.

2.4 Summary

Section 2.3 showed that chip yield drastically reduces from 85% at 250nm to 60% at 50nm, demonstrating that just having array redundancy in the caches will be insufficient at future technologies. Traditionally, to prevent lower yields as we move to newer process technologies fabrication plants put substantial effort into the reduction of baseline defect densities and prevention of defect excursions. As we move to smaller device dimensions and larger number of devices per chip, it is questionable whether a methodology based on defect avoid-

ance or precise defect detection would scale well enough to provide us the maximum yield we can hope to achieve at that technology generation.

DRAM and on-chip storage structures such as caches have long taken advantage of their regular structure by supplying redundant rows and columns which could be switched over to take the place of bad cells to increase the chip yield. Many structures in modern chip design, beyond just DRAM and SRAM, already contain some degree of regularity. Motivated by technology trends such as wire delays, limits on deep pipelining, and power consumption, several researchers have proposed architectures which employ extensive partitioning, regularity, and adaptivity in all major microarchitectural components [34]. Multi-banked caches and register files have been proposed and implemented in current day processors to achieve higher bandwidth and lower access times. Recent work in designing large on-chip L2 caches makes use of a large number of banks connected by a 2D mesh network and simple routers/switches [49]. Proposals for future wire-delay scalable microprocessor architectures like the Grid Processor Architecture use a regular array of functional units connected by a light-weight operand network to achieve high parallelism and low operand latencies [50]. Research in power-aware microarchitectures have proposed fine grained adaptivity in the sizes of on-chip structures and execution resources to save dynamic power. Chip multiprocessor architectures are becoming popular providing opportunities for hierarchical redundancy within the chip.

Based on these architectural trends, we advocate pushing fail-in-place strategies inside the boundaries of a single chip or processor, by exploiting the microarchitectural redundancy available in modern and future designs. In Chapter 4, we will discuss the salient features of future distributed architectures that provide the foundation for building mechanisms for hard error reliability. Building on these features, we will discuss both hardware and software based hard error reliability enhancement techniques in Chapter 5.

Chapter 3

Impact of Technology Trends on Soft Error Rate

Two important drivers of microprocessor performance have been scaling of device feature sizes, and increasing pipeline depths. In this chapter, we explore how these trends affect the susceptibility of microprocessors to soft errors. Device scaling is the reduction in feature size and voltage levels of the transistors, which improves performance because smaller devices require less current to turn on or off, and thus can be operated at higher frequencies. Pipelining is a microarchitectural technique of dividing instruction processing into stages which can operate concurrently on different instructions. Pipelining improves performance by increasing instruction level parallelism (ILP). Five to eight stage pipelines are quite common, and modern processor designs use between 12-20 pipeline stages [36, 51]. Such designs are commonly referred to as *superpipelined* designs.

Our study focuses on *soft errors*, which are also called transient faults or single-event upsets (SEUs). These are errors in processor execution that are due to electrical noise or external radiation rather than design or manufacturing defects. Transient faults can arise from multiple sources: external radiation, capacitive coupling, leakage, power supply noise, and temporal circuit variations. In particular, we study soft errors caused by high-energy

neutrons resulting from cosmic rays colliding with particles in the atmosphere. The existence of cosmic ray radiation has been known for over 50 years, and the capacity for this radiation to create transient faults in semiconductor circuits has been studied since the early 1980s. As a result, most modern microprocessors already incorporate mechanisms for detecting soft errors. These mechanisms are typically focused on protecting memory elements, particularly caches, using error-correcting codes (ECC), parity, and other techniques. Two key reasons for this focus on memory elements are: 1) the techniques for protecting memory elements are well understood and relatively inexpensive in terms of the extra circuitry required, and 2) caches take up a large part, and in some cases a majority, of the chip area in modern microprocessors.

Past research has shown that combinational logic is much less susceptible to soft errors than memory elements [52, 53]. Three phenomena provide combinational logic a form of natural resistance to soft errors: 1) logical masking, 2) electrical masking, and 3) latching-window masking. We develop models for electrical masking and latching-window masking to determine how these are affected by device scaling and superpipelining. Then based on a composite model we estimate the effects of these technology trends on the soft error rate (SER) of combinational logic. Finally, using an overall chip area model we compare the SER/chip of combinational logic with the expected trends in SER of memory elements.

The primary contribution of this study is an analysis of the trends in SER for SRAM cells, latches, and combinational logic. Our models predict that by the 50nm technology generation, the raw soft error rate of combinational logic will be comparable to that of latches, and will be within two orders of magnitude of unprotected memory elements. This result is significant because current methods for protecting combinational logic from soft errors have significant costs in terms of chip area, performance, and/or power consumption in comparison to protection mechanisms for memory elements.

The rest of this chapter is organized as follows. Section 3.1 provides background on

the nature of soft errors, and a method for estimating the soft error rate of memory circuits. Section 3.2 introduces our definition of soft errors in combinational logic, and examines the phenomena that can mask soft errors in combinational logic. Section 3.3 describes in detail our methodology for estimating the soft error rate in combinational logic. We present our results in Section 3.4. Section 3.5 discusses the implications of our analysis and simulations. Section 3.6 summarizes the related work, and Section 3.7 concludes the chapter with ideas for improving the soft error reliability of future processors.

3.1 Background

3.1.1 Particles that Cause Soft Errors

Cosmic rays are particles that originate from outer space and enter the earth's atmosphere. These particles may collide with other particles in the atmosphere, which may in turn be accelerated toward earth. A measure of this form of radiation is the flux, or rate of flow, expressed as the number of particles passing through a a given area per unit of time. The final flux of particles that reaches a location on the earth depends on a number of factors, including:

- **Altitude:** Higher altitudes see higher rates of particles. The flux at an altitude of 3100m (Leadville, CO) is approximately 13 times greater than at sea level.
- **Geomagnetic region (GMR):** This factor relates to the shielding from cosmic rays that results from the magnetic field around the earth. This shielding effect is strongest around the equator and weakest at the poles. GMR is a measure of this shielding effect, and is expressed in units of volts. Measurements of GMR have been performed at various locations on the earth, and these measurements range from 1.0 GV near the poles to as high as 17 GV at the equator.
- **Solar cycle:** Particle flux is also affected by the eleven-year solar cycle. Periods

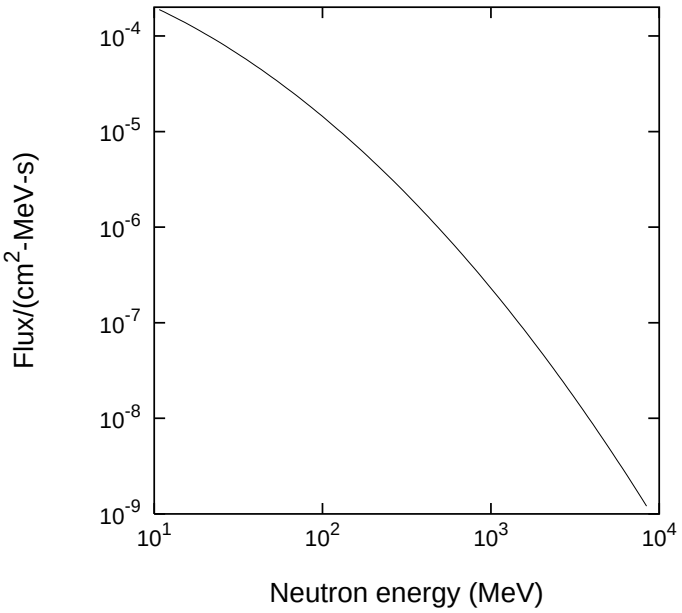

Figure 3.1: Particle flux: Energy distribution of neutron flux from cosmic rays, with energy level on the x-axis and the flux on the y-axis. The most important aspect is that, particles of lower energy occur far more frequently than particles of higher energy.

of active sun see up to a 30% lower rate of particles compared to periods of quiet sun. This is somewhat contrary to the common belief that an active sun increases the flux of cosmic particles. In fact, the magnetic field around the earth strengthens during periods of active sun, increasing the shielding effect and thus reducing cosmic particle flux. Solar flares can temporarily generate increased particle rates, but the increase in magnetic shielding during the active sun period outweighs these events.

In the early 1980s, IBM conducted a series of experiments to measure the particle flux from cosmic rays [54]. The graph in Figure 3.1 presents their findings. This graph shows the energy distribution of the neutron flux, where the energy level is given on the x-axis and the flux is shown on the y-axis. Only neutrons are shown in the graph since they account for more than 97% of the cosmic particles to reach sea level [55]. The data is normalized to a sea level location with a GMR of 1.2GV in 1985 (quiet sun period).

The total flux of particles greater than 10MeV is 0.00565/cm^2-s. For this study, the most important aspect of these results is that particles of lower energy occur far more frequently than particles of higher energy. In particular, a one order of magnitude difference in energy can correspond to a two orders of magnitude larger flux for the lower energy particles. As CMOS device sizes decrease, they can be affected by particles with lower energy levels, potentially leading to a much higher rate of soft errors.

This chapter investigates the soft error rate of combinational logic caused by atmospheric neutrons with energies greater than 1 mega-electron-volt (MeV). This form of radiation, the result of cosmic rays colliding with particles in the atmosphere, is known to be a significant source of soft errors in memory elements. We do not consider atmospheric neutrons with energy less than 1 MeV since we believe their much lower energies are less likely to result in soft errors in combinational logic. We also do not consider alpha particles, since this form of radiation comes almost entirely from impurities in packaging material, and thus can vary widely for processors within a particular technology generation. The contribution to the overall soft error rate from each of these other radiation sources is additive, and thus each component can be studied independently.

3.1.2 Soft Errors in Memory Circuits

High-energy neutrons that strike a sensitive region in a semiconductor device deposit a dense track of electron-hole pairs as they pass through a p-n junction. Some of the deposited charge will recombine to form a very short duration pulse of current at the internal circuit node that was struck by the particle. The magnitude of the collected charge depends on the particle type, physical properties of the device, and the circuit topology. When a particle strikes a sensitive region of an SRAM cell, the charge that accumulates could be large enough to flip the value stored in the cell, resulting in a soft error. The smallest charge that results in a soft error is called the *critical charge* (Q_{CRIT}) of the SRAM cell [56]. The rate at which soft errors occur is typically expressed in terms of *Failures In Time (FIT)*, which

measures the number of failures per 10^9 hours of operation. A number of studies on soft errors in SRAMs have concluded that the SER for constant area SRAM arrays will increase as device sizes decrease [57–59], though researchers differ on the rate of this increase.

A method for estimating SER in CMOS SRAM circuits was recently developed by Hazucha & Svensson [60]. This model estimates SER due to atmospheric neutrons (neutrons with energies $>$ 1MeV) for a range of submicron feature sizes. It is based on a verified empirical model for the 600nm technology, which is then scaled to other technology generations. The basic form of this model is:

$$SER = K \times F \times A \times \exp\left(-\frac{Q_{CRIT}}{Q_S}\right) \tag{3.1}$$

where

K	is a constant independent of device technology with the value $2.2 * 10^{-5}$,
F	is the neutron flux with energy $>$ 1 MeV, in particles/(cm^2·s),
A	is the area of the circuit sensitive to particle strikes, in cm^2,
Q_{CRIT}	is the critical charge, in femto Coulomb (fC), and
Q_S	is the charge collection efficiency of the device, in fC

Two key parameters in this model are the critical charge (Q_{CRIT}) of the SRAM cell, and the charge collection efficiency (Q_S) of the circuit. Q_{CRIT} depends on characteristics of the circuit, particularly the supply voltage and the effective capacitance of the drain nodes. Q_S is a measure of the magnitude of charge generated by a particle strike. These two parameters are essentially independent, but both decrease with decreasing feature size. From Equation 3.1 we see that changes in the value of Q_{CRIT} relative to Q_S will have a very large impact on the resulting SER. The SER is also proportional to the area of the sensitive region of the device, and therefore it decreases proportional to the square of the device size. Hazucha & Svensson used this model to evaluate the effect of device scaling on

the SER of memory circuits. They concluded that SER-per-chip of SRAM circuits should increase at most linearly with decreasing feature size.

3.2 Soft Errors in Combinational Logic

A particle that strikes a p-n junction within a combinational logic circuit can alter the value produced by the circuit. However, a transient change in the value of a logic circuit will not affect the results of a computation unless it is captured in a memory circuit. Therefore, we define a soft error in combinational logic as a transient error in the result of a logic circuit that is subsequently stored in a memory circuit of the processor.

A transient error in a logic circuit might not be captured in a memory circuit because it could be *masked* by one of the following three phenomena:

Logical masking occurs when a particle strikes a portion of the combinational logic that is blocked from affecting the output due to a subsequent gate whose result is completely determined by its other input values.

Electrical masking occurs when the pulse resulting from a particle strike is attenuated by subsequent logic gates due to the electrical properties of the gates to the point that it does not affect the result of the circuit.

Latching-window masking occurs when the pulse resulting from a particle strike reaches a latch, but not at the clock transition where the latch captures its input value.

These masking effects have been found to result in a significantly lower rate of soft errors in combinational logic compared to storage circuits in equivalent device technology [53]. However, these effects could diminish significantly as feature sizes decrease and the number of stages in the processor pipeline increases. Electrical masking could be reduced by device scaling because smaller transistors are faster and therefore may have less attenuation effect

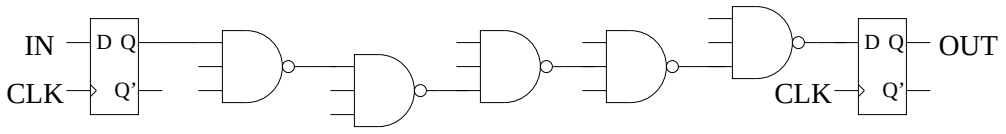

Figure 3.2: Simple model of a pipeline stage used as the primitive circuit for estimating SER of combinational logic

on a pulse. Also, deeper processor pipelines allow higher clock rates, meaning the latches in the processor will cycle more frequently, which may reduce latching-window masking.

The direct consequences of device scaling are that the supply voltage, threshold voltages and the dimensions of the device decrease. These have the effects of reduced ON currents in the transistors, decreased values of node capacitances and a lower input to output delay. Since the ON currents of the device are lower the transient current pulses due to particle strikes are of comparable magnitude making the device less stable. Since the transistors are now faster, an input pulse's rise time, fall time, and width don't degrade significantly as the pulse propagates through the gates, increasing the chances of an error pulse successfully propagating to the latch. So device scaling trends tend to make the transistors more susceptible to soft errors.

The datapath of modern processors can be extremely complicated in nature, typically composed of 64 parallel bit lines, and divided into 12-20 pipeline stages. We evaluate the effects of electrical and latching-window masking using the simple model for a processor pipeline stage illustrated in Figure 3.2. This model is just a one-wide chain of homogeneous gates terminating in a level-sensitive latch. For the results presented in this chapter we use static 3-input NAND gates with a fan-out of 4.

The number of gates in the chain is determined by the degree of pipelining in the microarchitecture, given as the number of fan-out-of-4 inverter (FO4) gates that can be placed between two latches in a single pipeline stage. We use the FO4 metric because it allows us to characterize pipeline depth in a way that is largely independent of device scaling [33]. During the last sixteen years, technology has scaled from 1000nm to 65nm

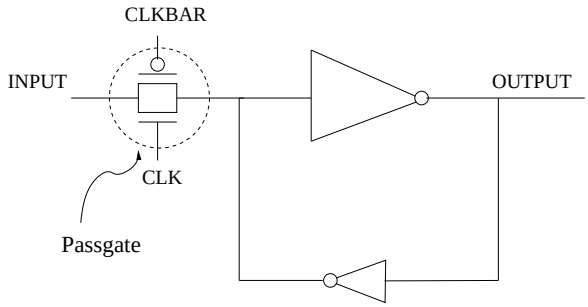

Figure 3.3: Circuit diagram of a pipeline latch

and the amount of logic per pipeline stage has decreased from 84 to 12 FO4 contributing to a total of 60-fold increase in clock frequency in the Intel family of processors. For a given degree of pipelining, the number of gates in the pipeline stage is the largest number that does not exceed the total delay of the corresponding FO4 chain.

Figure 3.3 shows the circuit diagram of the latch we used in our simple pipeline model. The forward inverter is about 6 times larger than the feedback inverter and the transistors are all of minimum length. We use level sensitive latches in our pipeline model because they occupy less area than edge triggered flip-flops and so are more suitable for superpipelining. They also allow for time borrowing techniques and offer less load to the clock distribution network thus reducing the clock skew in the chip.

We used 3-input NAND gates with a fan-out of 4 because they are a common gate used in many logic designs and result in a conservative estimate of chip SER. The critical charge of a circuit increases with the capacitance associated with it. For example, the output node of a 3-input NAND gate has a much larger capacitance than an inverter with the same drive strength and so has a greater critical charge. By the same reasoning, a NAND gate with a fan-out of 4 has a greater critical charge than a NAND gate that only drives a single following gate.

3.3 Methodology

In most modern microprocessors, combinational logic and memory elements are constructed from the same basic devices – NMOS and PMOS transistors. Therefore, we can use techniques for estimating the SER in memory elements to assess soft errors in combinational logic. We will also use these techniques directly to compute the SER in memory elements for a range of device sizes, and compare the results to our estimates of SER for combinational logic.

Our methodology for estimating the soft error rate in combinational logic considers the effects of CMOS device scaling, and the microarchitectural technique of processor pipelining. We determine the soft error rate using analytical models for each stage of the pulse from its creation to the time it reaches the latch. Figure 3.4 shows the various stages the pulse passes through and the corresponding model used to determine the effect on the pulse at that stage. In the first stage, the charge generated by the particle strike produces a current pulse, which is then converted into a voltage pulse after traveling through a gate in the logic chain. The electrical masking model simulates the degradation of the pulse as it travels through the gates of the logic circuit. Finally, a model for the latching window determines the probability that the pulse is successfully latched. The remainder of this section describes each of these component models and how they are combined to obtain an estimate for the SER of combinational logic.

3.3.1 Device Scaling Model

We constructed a set of Spice Level 3 technology models corresponding to the technology generations from the Semiconductor Industry Association 1999 Technology Roadmap [38]. Values for drawn gate length (L_{DRAWN}), supply voltage (V_{DD}), and oxide thickness (TOX) are taken directly from the roadmap. With the exception of threshold voltage (V_{TH}), the remaining parameters were obtained using a scaling methodology developed by McFarland [61]. We chose a slightly different formula for computing the threshold voltage which

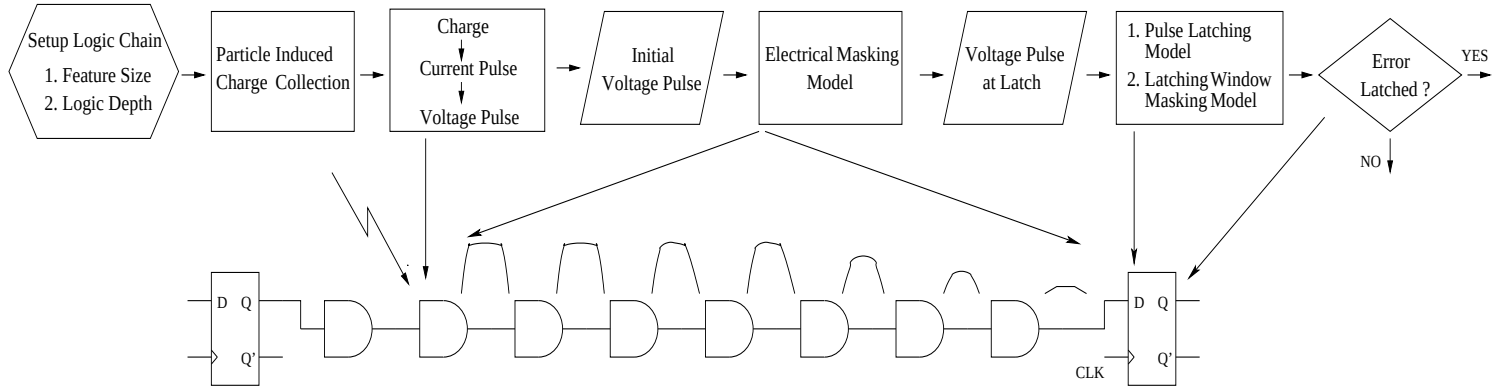

Figure 3.4: Process for determining the Soft Error Rate in a logic chain

Technology Generation	600nm	350nm	250nm	180nm	130nm	100nm	70nm	50nm
L_{DRAWN} (nm)	600	350	250	140	90	65	45	32
V_{DD} (V)	5.0	3.3	2.5	1.8	1.5	1.2	0.9	0.6
TOX (nm)	11	7.6	4.0	2.5	1.9	1.5	1.2	0.8
V_{TH} (V)	1.0	0.735	0.596	0.466	0.407	0.344	0.277	0.205

Table 3.1: Key characteristics of CMOS device models

scales better to technologies with very low supply voltages. In the McFarland model, V_{TH} was set to $0.40 \times V_{DD}^{0.56} + 0.2$. The constant term of 0.2 volts led to poor scaling for small values of V_{DD}, so instead we use the formula $V_{TH} = 0.30 \times V_{DD}^{0.75}$. Table 3.1 presents the key characteristics of our CMOS device models. The supply voltages in our model roughly scale by 0.7-0.8X every generation, and track the published voltage values for the Intel family of processors quite well. Modern processor designs however have multiple versions with different supply voltages, and often come with the capability to dynamically adjust the supply voltage to be able to configure the power-performance tradeoff. We analyze one supply voltage at each generation, while recognizing that the SER depends on the supply voltage used.

3.3.2 Charge to Voltage Pulse Model

When a particle strikes a sensitive region of a circuit element it produces a current pulse with a rapid rise time, but a more gradual fall time. The shape of the pulse can be approximated by a one-parameter function shown in Equation 3.2 [56].

$$I(t) \propto \frac{Q}{T} \times \sqrt{\frac{t}{T}} \times \exp\left(-\frac{t}{T}\right) \tag{3.2}$$

Q refers to the amount of charge collected due to the particle strike. The parameter T is the time constant, in units of nanoseconds, for the charge collection process and is a property of the CMOS process used for the device. If T is large, it takes more time for

the charge to recombine. If T is small, the charge recombines rapidly, generating a current pulse with a short duration. The rapid rise of the current pulse is captured in the square root function and the gradual fall of the current pulse is produced by the negative exponential dependence. Figure 3.5 illustrates the pulse waveform generated by this equation for the 100nm technology generation and a charge value $Q = 100$ fC.

The time constant T scales approximately linearly with feature size in a natural log-log scale [62]. We constructed a model for T for any CMOS technology, characterized by the minimum gate length g (in μm), by fitting a straight line through the values of T for 600nm, 350nm and 100nm from [60]. This model is given in Equations 3.3 and 3.4. The curve fitting was done using Matlab and the correlation coefficients were high enough for the errors to be insignificant.

$$\text{NMOS: } T = \exp(0.97 \times \ln(g) + 5.5) \tag{3.3}$$

$$\text{PMOS: } T = \exp(0.81 \times \ln(g) + 5.2) \tag{3.4}$$

where g is specified in μm, and the resulting T value is in picoseconds.

The current pulse produced by a particle strike results in a voltage pulse at the output node of the device. We use a Spice simulation to determine the rise time, fall time and effective duration of this voltage pulse. The effective duration is the elapsed time the pulse exceeds half the supply voltage. These three values are the final result of this stage and become the input for the next phase, the electrical masking analytical model.

3.3.3 Electrical Masking Model

Electrical masking is the composition of two electrical effects that reduce the strength of a pulse as it passes through a logic gate. Circuit delays caused by the switching time of

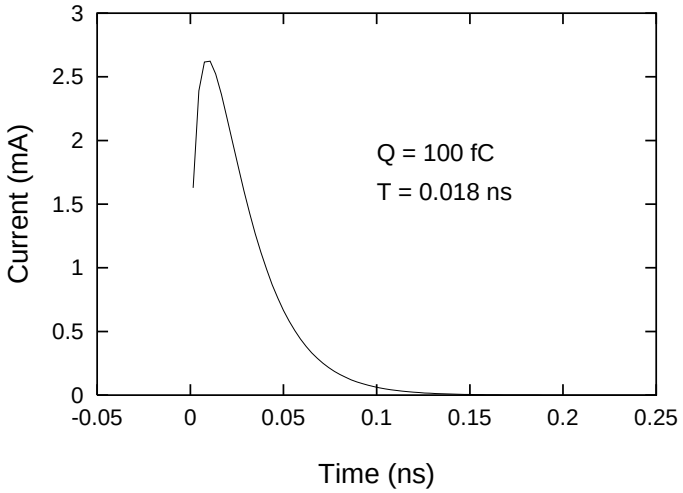

Figure 3.5: A current pulse resulting from a particle strike at a device node

the transistors cause the rise and fall time of the pulse to increase, reducing its effective duration. For short duration pulses, pulse duration is further reduced because the gate may start to turn off before the output reaches its full amplitude. As pulse duration decreases, this second effect becomes greater, and thus these effects cascade from one gate to the next, and can eventually degrade the pulse to the extent that it cannot affect the result latch.

We define the *rise time* of a pulse to be the time for the pulse to rise from GND to V_{DD}. For pulses that do not actually rise all the way to V_{DD}, we extend the rising edge and measure rise time to the point where this edge crosses V_{DD}. *Fall time* is defined similarly. Using these definitions, rise time and fall time are best thought of as describing the slope of the rising and falling edge. For the experiments reported in this chapter, we model a pipeline stage as a chain of static, 3-input, fan-out-of-4 NAND gates. One output of each gate feeds one input of the next gate in the chain, with the other inputs fixed at a logical 1. In this model, each gate in the chain inverts the signal on its one non-fixed input, so a rising pulse entering the gate becomes a falling pulse leaving the gate, and vice-versa.

We constructed a model for electrical masking based on the propagation delay of

an electrical signal through a logic gate. Gate delay is determined using a composition of two existing models. The Horowitz gate delay model [63] determines the normal gate delay based on the rise or fall time of the input signal and the gate switching voltage. This normal gate delay is then adjusted to account for short duration input signals using a model by Bellido-Diaz *et al.* [64].

Determining gate delay

In the Horowitz gate delay model, the delay of a gate is defined as the time between the input reaching the switching voltage of the gate and the output reaching the switching voltage of the following gate. The switching voltage of a gate determines the point at which the output of the gate is affected by the input(s) to the gate. We use a form of the Horowitz gate delay model that allows the switching gate and the following gate to have different switching voltages, as described in [65]. This model is given by the following equations for a rising and falling input:

$$\begin{aligned} delay_{rise} &= t_f \cdot \sqrt{\left[\log\left(\frac{V_{TH1}}{V_{DD}}\right)\right]^2 + \frac{2\,t_{rise}\,b}{t_f}\left(1-\frac{V_{TH1}}{V_{DD}}\right)} \\ &\quad + t_f\left[\log\left(\frac{V_{TH1}}{V_{DD}}\right) - \log\left(\frac{V_{TH2}}{V_{DD}}\right)\right] \end{aligned}$$

$$\begin{aligned} delay_{fall} &= t_f \cdot \sqrt{\left[\log\left(1-\frac{V_{TH1}}{V_{DD}}\right)\right]^2 + \frac{2\,t_{fall}\,b}{t_f}\left(\frac{V_{TH1}}{V_{DD}}\right)} \\ &\quad + t_f\left[\log\left(1-\frac{V_{TH1}}{V_{DD}}\right) - \log\left(1-\frac{V_{TH2}}{V_{DD}}\right)\right] \end{aligned}$$

where

t_f is the output time constant (assuming a step input),

t_{rise} is the rise time of the input signal,

t_{fall} is the fall time of the input signal,

V_{TH1} is the switching voltage of the switching gate,

V_{TH2} is the switching voltage of the following gate,

b is the fraction of the swing in which the input affects the output (we used $b = 0.5$ for rising inputs and $b = 0.4$ for falling inputs).

We calibrated the Horowitz gate delay model to our CMOS design parameters by adjusting the gate switching voltage parameters to achieve close agreement between the model outputs and corresponding Spice simulation results. The gate switching voltages are determined using an iterative bisection method. This procedure adjusts the switching voltages until the rise and fall times predicted by the model are within 15% of values obtained from Spice simulations. In general, the values obtained from this procedure differ from the actual gate switching voltages, which can be determined by measurements in Spice. Nevertheless, calibration significantly improves the rise and fall time estimates of the model, so we chose to treat switching voltages as "one degree of freedom" for the sake of improved accuracy. Table 3.2 shows the switching voltages (normalized to V_{DD} for each technology) determined using this procedure for the NAND gate used in the experiments. To calculate the delay for a rising edge, we set V_{TH1} to V_{rise} and V_{TH2} to V_{fall}; for a falling input, V_{TH1} is set to V_{fall} and V_{TH2} to V_{rise}.

Technology Generation	600nm	350nm	250nm	180nm	130nm	100nm	70nm	50nm
V_{rise}/V_{DD}	0.16	0.50	0.38	0.31	0.31	0.38	0.38	0.38
V_{fall}/V_{DD}	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25

Table 3.2: Switching voltage of the gates for the rise and the falling transitions at different process technologies

The pulse generated by a particle strike typically has a short duration. Thus, when a pulse passes through a logic gate, the output of the gate might not have time to fully switch

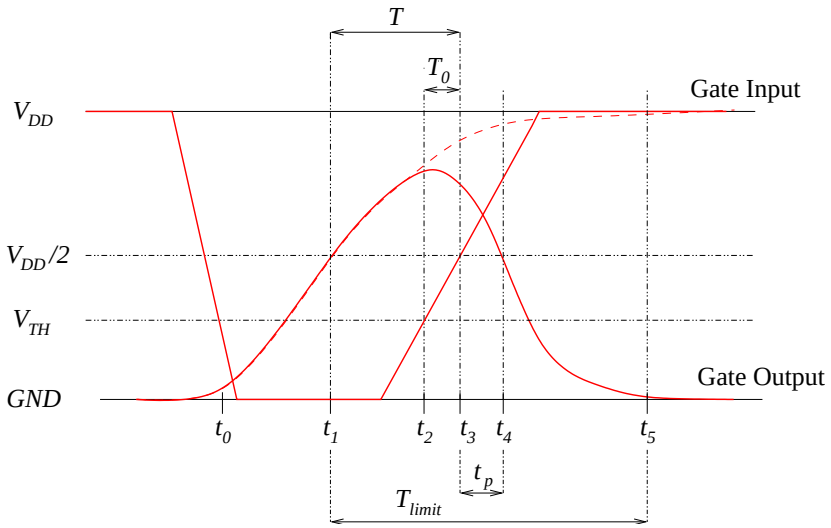

Figure 3.6: The delay degradation effect

in response to the pulse before it disappears from the gate input. If the output has not fully switched, the gate can respond to the new state of its inputs more quickly, and thus the gate delay is reduced. This effect is know as the *delay degradation effect* [64]. In this situation, the value generated at the gate output begins switching in the opposite direction before reaching the peak amplitude of the input, which results in an output signal with reduced amplitude. We use a model by Bellido-Diaz *et al.* to simulate this effect on an error pulse as it passes through a logic gate.

Figure 3.6 illustrates the delay degradation effect. At time t_0, the gate input crosses the gate switching voltage, V_{TH}, and the gate output starts to rise. At time t_1 the gate output crosses $V_{DD}/2$, and thus has logically transitioned to the new output value. Then the gate input begins rising and crosses back above the gate switching voltage at time t_2, and as a result, the gate output begins to fall before it could rise completely to V_{DD}. Since the output did not reach it's full amplitude, it requires less time to fall back below $V_{DD}/2$, resulting in a smaller than normal propagation delay t_p. If the input had remained low until time t_4, the output would rise fully to V_{DD}, and a subsequent input transition will result in a normal

propagation delay. The Bellido-Diaz delay degradation model is given in Equation 3.5.

$$t_p = t_{p0}\left(1 - e^{\frac{T_0 - T}{\tau}}\right) \qquad \text{where} \qquad \tau = \frac{T_{limit}}{3} \tag{3.5}$$

t_{p0} is the normal propagation delay with zero degradation effect, which we determine using the Horowitz model. The rest of the equation captures the degradation effect. The time between the output transition and the next input transition is (T-T_0), and τ is a parameter proportional to the time needed for the gate to fully switch, T_{limit}.

Determining output pulse characteristics from gate delay

We approximate the input and output of the gate as piecewise-linear signals, and use a simple linear model to determine the rise and fall time of the gate output based on rise and fall time of the input and the gate delay. Figure 3.7 illustrates the model for the fall time of the gate output. In this figure, the rising edge of the pulse is passing through a gate, causing the output to fall from V_{DD} to GND. Thus, the delay of a gate also determines the time required for the output signal to rise(fall) to the switching voltage of the next gate. Using a simple linear model, we can extend this rising(falling) edge to V_{DD}(GND) to determine the rise/fall time of the output pulse. Figure 3.7 illustrates this calculation. The rise time for this rising edge is t_{rise}. When the input pulse crosses V_{TH1}, the switching voltage of the gate, the output begins to fall. The output reaches the V_{TH2}, the switching voltage of the following gate, after time $delay_{rise}$, the gate delay for the rising edge of the pulse. Thus we can determine the base and height of triangle $a\,b\,c$. The fall time of the output pulse, t_{fall}, is the base of the equivalent triangle $a\,d\,e$, whose height is simply V_{DD}. Thus, t_{fall} is calculated as follows:

$$t_{fall} = delay_{rise} \Big/ \left(1 - \frac{V_{TH2}}{V_{DD}}\right) \tag{3.6}$$

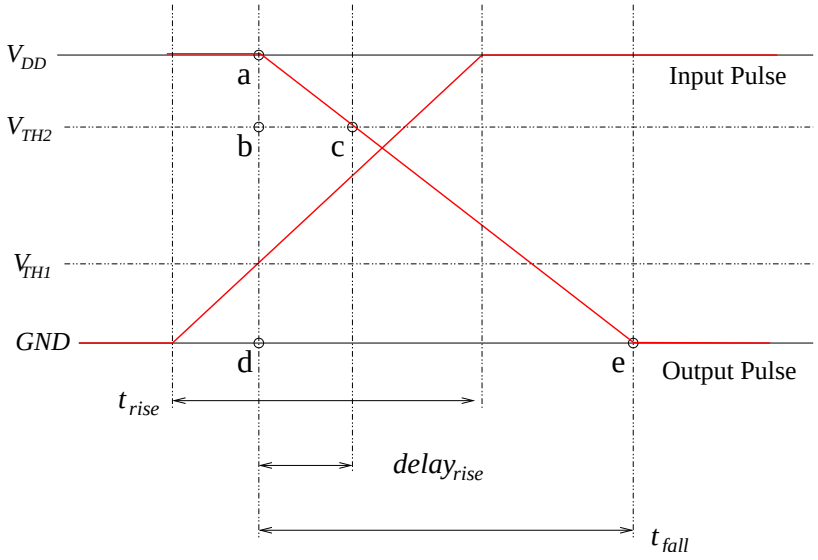

Figure 3.7: Model to compute rise/fall time based on gate delay

We assume that the input to the gate is stable when the pulse arrives, and thus there is no delay degradation for the leading edge (which could be either a rising or falling edge). However, when the pulse is short, the gate delay for trailing edge could be significantly smaller than that of the leading edge, and this reduces the duration of the output pulse. Thus, we determine the duration of the output pulse as

$$duration_{output} = duration_{input} - (delay_{leading} - delay_{trailing}) \tag{3.7}$$

3.3.4 Pulse Latching Model

Recall that our definition of a soft error in combinational logic requires an error pulse to be captured in a memory circuit. Therefore, in our model a soft error occurs when the error pulse is stored into the level-sensitive latch at the end of a logic chain. We only consider a value to be stored in the latch if it is present and stable when the latch closes, since this value is passed to the next pipeline stage.

When a voltage pulse reaches the input of a latch, we use a Spice simulation to determine if it has sufficient amplitude and duration to be captured by the latch. The simulation is done in two steps. First we determine the pulse start time, the shortest time between the rising edge of the pulse and clock edge in which the pulse could be latched. This is similar to a setup time analysis for the latch, except that the input data waveform has the slope of the pulse at the latch input. The second step is to determine the minimum duration pulse (measured at the threshold voltage) that could be latched. For this step, we position the rising edge of the pulse at the point determined in the first step, and then vary the duration until the minimum value is determined. We studied the nature of the pulse start time and minimum duration using separate experiments and found that the pulse start time can be modeled by a linear function of the rise time of the pulse, and the minimum duration can be modeled by a linear function of the rise time and fall time. For example, the pulse start time (in ps) of our pipeline latch in our 600nm technology can be computed as follows:

$$\text{start} = 65.8 + 0.375 \times t_{rise}$$

and the minimum duration (in ps) is given by

$$\text{duration} = 106 + 0.323 \times t_{rise} + 0.448 \times t_{fall}$$

In our method for computing SER for combinational circuits, the start time and minimum duration of an error pulse must be computed very frequently. Therefore, it is

important that we determine these values using a simple model rather than with Spice simulations so that run times for the overall model are reasonable. The pulse start time and minimum duration given by these models correlate very highly with the pulse start time and minimum duration determined from Spice simulations, and therefore allow us to replace an expensive simulation with a very inexpensive calculation without significant loss in accuracy.

Given the rise and fall time of a pulse at the latch input, the simulation determines the minimum duration (measured at the threshold voltage) required for the pulse to be latched. If the duration of the pulse at the latch input exceeds this minimum duration, the pulse has the potential to cause a soft error. This method determines if a particle-induced pulse in an otherwise stable, correct input signal is strong enough to be latched. It is also possible that a particle-induced pulse could delay the correct input signal from arriving at the latch input in time to be latched, thus causing an error. This type of error is referred to as a *delay fault*. Due to the complexity of modeling these faults, we have chosen to exclude them from our study. Bernstein found that delay faults are negligible in current technologies due to the common design practice of incorporating a 5%-10% safety margin into the clock cycle [66]. However, such faults could become much more common as clock frequency increases and safety margins are squeezed to increase performance.

3.3.5 Latching-window Masking Model

A latch is only vulnerable to a soft error during a small window around its closing clock edge. The size of this *latching window* is simply the minimum duration pulse that can be latched, which depends on the pulse rise and fall time. A pulse that is present at the latch input throughout the entire latching window will be latched and causes a soft error. In our model, any pulse with a duration smaller than the duration of the latching window cannot cause a soft error. Figure 3.8 illustrates our model of latching window masking. Only a pulse that completely overlaps the latching window results in a soft error. If the pulse either

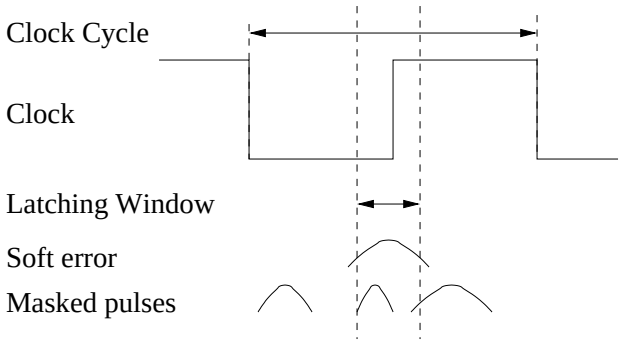

Figure 3.8: Latching-window masking

arrives after the latching window has opened, terminates before the latching window closes, or does not have sufficient duration to cover the whole window, we assume that the pulse will be masked.

Let d represent the duration of the pulse on arrival at the latch input at time t. The pulse arrival time t can occur at any point in the clock cycle with equal probability. Let w represent the size of the latching window for this pulse, and let c represent the clock cycle time. If a latching window for the latch starts after time t and ends before time $t + d$, the pulse is present at the latch input throughout the entire latching window and results in a soft error. Otherwise the pulse is masked and no soft error occurs.

We can determine the probability that the pulse causes a soft error by computing the probability that a randomly placed interval of length d overlaps a fixed interval of length w within an overall interval of length c. This probability is given by the following equation:

$$\Pr\text{soft error} = \begin{cases} 0 & \text{if } d < w \\ \frac{d-w}{c} & \text{if } w \leq d \leq c + w \\ 1 & \text{if } d > c + w \end{cases} \tag{3.8}$$

Note that when $d < w$, the probability of a soft error is zero, but this is not an effect of latching window masking, since the pulse does not have sufficient duration to be

latched. On the other hand, when the pulse duration exceeds $c+w$, it is assured to overlap at least one full latching window of size w and hence has probability 1 of causing a soft error. Note that a smaller pulse could partially overlap the latching windows in two consecutive clock cycles without fully containing either one. Since pulse arrival times are distributed uniformly at random over the clock cycle, the probability of an error for a pulse with any intermediate duration is a simple linear function between these two endpoints.

3.3.6 Estimating SER of Combinational Logic

We assume that the probability of concurrent particle strikes in a single logic chain is negligible, and thus the SER for the circuit is simply the sum of the SER's for a particle strike at each gate in the logic chain. To compute the SER contribution for a given gate in the logic chain, we simulate a particle strike to the drain of the gate using our charge to voltage pulse model. Then we apply our electrical masking model to determine the characteristics of the voltage pulse when it reaches the latch input. We use the pulse-latching model to determine if the pulse that reaches the latch input has sufficient amplitude and duration to cause a soft error. As in memory circuits, the smallest charge that can generate a pulse that results in a soft error is the critical charge (Q_{CRIT}) for the circuit. In memory circuits, soft errors are essentially deterministic, in that no charge less that Q_{CRIT} can cause a soft error, and every charge of Q_{CRIT} or larger results in a soft error with probability 1.0. In combinational logic, we need to consider the probability of latching-window masking when computing SER. This is done by considering a range of charge values. The lower bound of this range is Q_{CRIT}, and the upper bound of the range is Q_{CMAX}, which is the smallest charge that has probability of 1.0 of being latched according to our latching-window masking model, or which has a probability within epsilon of all greater charge values. Charge values between Q_{CRIT} and Q_{CMAX} have the potential to be masked by latching-window masking, but charge values of Q_{CMAX} or greater always result in a soft error.

To complete the calculation of SER for a given gate in the logic chain, we divide the

charge values between Q_{CRIT} and Q_{CMAX} into m equal-size intervals. We used $m = 20$ for the results presented in this chapter; using separate experiments we validated that using a higher granularity has only a marginal effect on the resulting SER estimates. We compute the SER corresponding to each interval using the model of Hazucha & Svensson. All our experiments use a value for the neutron flux of $F = 0.00565$, corresponding to sea level in New York City.

The charge collection efficiency Q_S scales approximately linearly with feature size in a natural log-log scale [62]. We constructed a model for Q_S for any CMOS technology, characterized by the minimum gate length g (in μm), by fitting a straight line through the values of Q_S for 600nm, 350nm and 100nm from [62]. This model is given in Equations 3.9 and 3.10. The curve fitting was done using Matlab and the correlation coefficients were high enough for the errors to be insignificant.

$$\text{NMOS: } Q_S = \exp(0.77 \times \ln(g) + 4.3) \tag{3.9}$$

$$\text{PMOS: } Q_S = \exp(1.0 \times \ln(g) + 4.2) \tag{3.10}$$

where g is specified in μm, and the resulting Q_S value is in fC.

Since the Hazucha & Svensson model gives a cumulative SER value, we compute the SER for an interval by subtracting the SER of the right endpoint of the interval from that of the left. The SER for the interval is then weighted by the probability that a soft error occurs as given by our latching-window masking model. The contribution to SER for the gate is then the sum of the weighted SER's for each interval plus the SER for Q_{CMAX}. This calculation is summarized in Equation 3.11.

$$\text{SER} = \text{SER}(Q_{CMAX}) + \sum_{i=1}^{m} \Pr L_i \left(\text{SER}(L_i) - \text{SER}(R_i)\right) \tag{3.11}$$

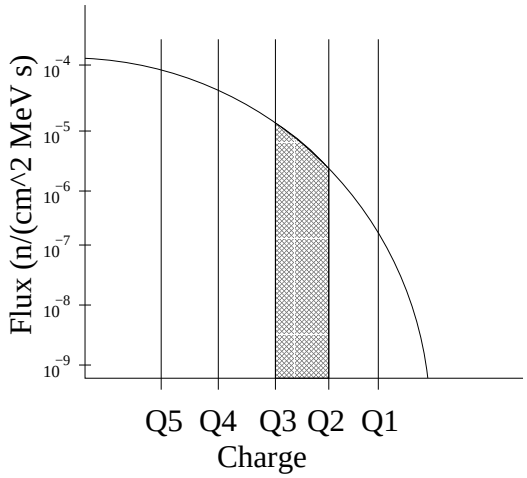

Figure 3.9: Computing SER using a range of charges with varying probability of latching.

where SER(Q) denotes the SER value for charge Q obtained from Hazucha & Svensson's model, L_i and R_i are the left and right endpoints of interval i, and $\Pr L_i$ is the probability that charge L_i causes a soft error (is not latching-window masked). This computation is illustrated in Figure 3.9. The contribution of the shaded region to overall SER is the SER for charges greater than Q_3 minus the SER for charges larger than Q_2, multiplied by the soft error probability associated with charge Q_3.

3.4 Results

3.4.1 Circuit Soft Error Rate

The circuits of a modern microprocessor fall into three basic classes: SRAM cells, latches, and combinational logic. We estimated the SER for an individual SRAM cell, latch, and logic chain using the methodology described in Section 3.3. Figure 3.10 shows the predicted SER by technology generation and pipeline depth. The x-axis plots the CMOS technology generation, arranged by actual or expected date of adoption. The y-axis plots the SER for a single SRAM cell, latch or logic chain, in *Failures In Time* (FIT) – the number of failures

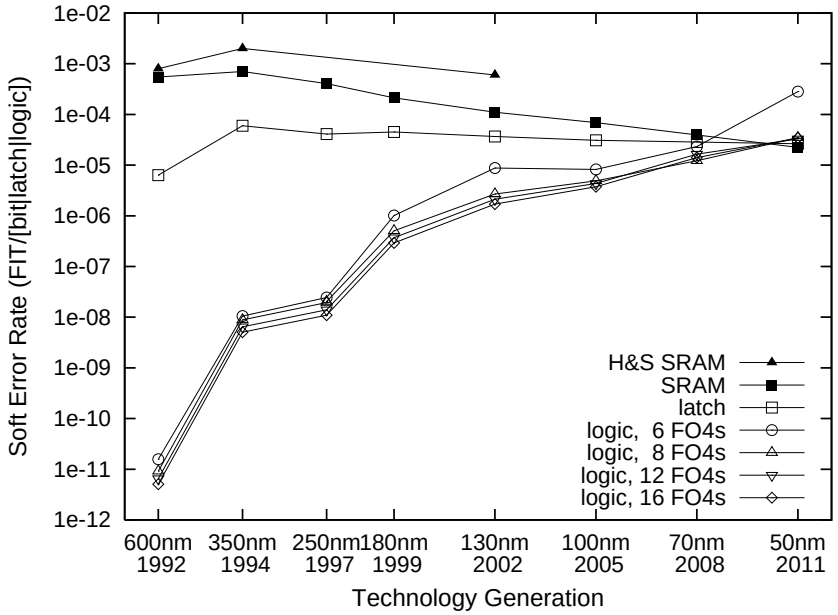

Figure 3.10: SER of individual circuits

per 10^9 hours of operation – on a log scale. Also shown in this graph are the results reported by Hazucha and Svensson for SRAM SER, using their scalable SER model [60].

The SER of a single SRAM cell declines gradually with decreasing feature size. There are three basic factors that combine to produce this trend. The drain area of each transistor, which is the region sensitive to particle strikes, decreases quadratically as feature size decreases. Critical charge also decreases significantly with decreasing feature size, primarily due to lower supply voltage levels, but also due to reduced capacitance in the smaller circuit nodes. Finally, charge accumulation in the transistor decreases due to reduced voltages and smaller node sensitive volume. In SRAM cells, the decrease in critical charge is effectively offset by reduced charge accumulation, and thus the decrease in sensitive area leads to a decrease in circuit SER. Our results show good correlation with those of Hazucha and Svensson; both results show the same basic trend, and the absolute error is less than one order of magnitude for all technologies, which can be attributed to differences in CMOS parameters.

The SER of a single latch stays relatively constant as feature size decreases. For latches, the effect of the decreasing area of the sensitive region is offset by the decrease in critical charge, as explained below. In contrast, the SER for a single logic chain changes dramatically as feature size decreases – increasing over six orders of magnitude from 600nm to 50nm. In logic circuits, the electrical effect of decreasing critical charge far outweighs the effect of decreasing area of the sensitive region. The effect of superpipelining is illustrated by the larger SER for logic circuits at higher pipeline depths (smaller clock period in FO4 delays) within each technology generation.

Decreasing critical charge: Recall that the empirical model for SER (Equation 3.1) has an exponential dependence on the ratio $-Q_{CRIT}/Q_S$. When this ratio is large, this factor dominates the SER expression, but its influence decreases rapidly as the value of Q_{CRIT} approaches Q_S. Figure 3.11 plots Q_{CRIT}, in femto-Coulombs, for an individual SRAM cell, latch, and logic circuit, along with Q_S, the charge collection efficiency, by technology generation. For combinational logic, the graph shows Q_{CRIT} values for a particle strike 0, 4, and 16 FO4 gate-delays from the latch. Note that the y-axis of the graph is log-scale. This data is also presented in Table 3.3. The values shown are for NMOS devices. Also, note that the data presented in Figure 3.11 differs somewhat from that contained in our earlier conference paper [67]. This is due to a minor problem in our technique for determining Q_{crit} which overstated Q_{crit} values whenever Q_{crit} was less than Q_S. Fortunately, this error has virtually no significant impact on the results shown in the rest of the paper.

For a single SRAM cell, Q_{CRIT} is only slightly larger than Q_S in the 350nm and 250nm technology generations, and falls below Q_S at 180nm. Even though Q_{CRIT} continues to fall as feature size decreases, the effect on SER is relatively small in comparison to the decreasing area of the sensitive region.

For a single pipeline latch, Q_{CRIT} is nearly an order of magnitude larger than Q_S in the 600nm technology generation, but declines steadily as feature size decreases, and should fall below Q_S by the 100nm technology generation. As Q_{CRIT} decreases relative

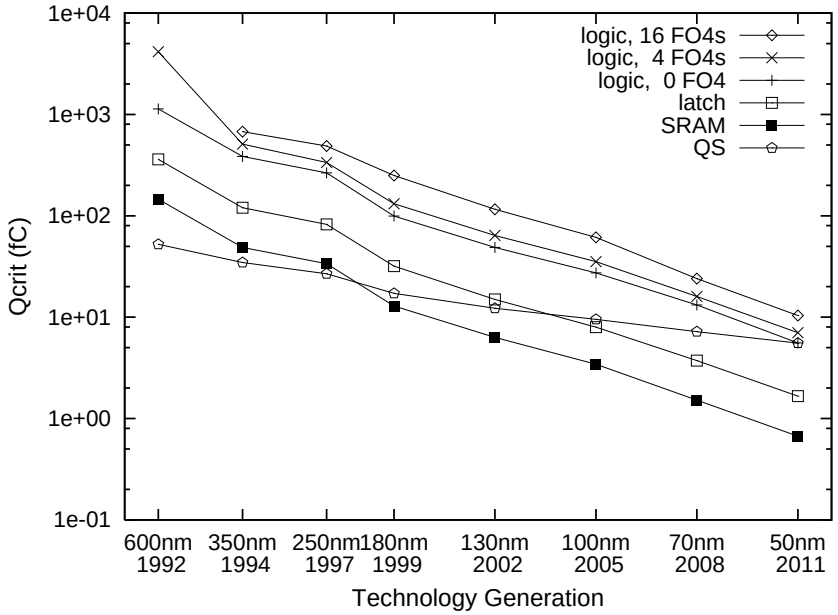

Figure 3.11: Critical charge for a single SRAM cell/latch/logic chain

to Q_S, the electrical effects of decreasing feature size diminish, and SER is more heavily influenced by the area of the sensitive region.

For a single logic chain, Q_{CRIT} decreases in a similar fashion to that of memory circuits, but at all points is much larger in absolute terms. Logic transistors are typically wider than transistors used in memory circuits, where density is important, and therefore are less sensitive to small charge values. Thus, the electrical effect of decreasing Q_{CRIT} is much larger than the area effect in all technology generations. Figure 3.11 also illustrates the effect of electrical masking on the SER of logic circuits. For all feature sizes below 600nm, the Q_{CRIT} for 16 FO4 logic gates is consistently about twice that of the 0 FO4 circuit, and this difference is the result of degradation of the error pulse as it passes through the 16 FO4 gates. Contrary to our expectations, our results do not show any reduction in this effect with decreasing feature size. We conclude that the primary effect of electrical masking is to screen out marginal pulses; the degradation effect on pulses with sufficient strength to be latched is minimal.

	600nm	350nm	250nm	180nm	130nm	100nm	70nm	50nm
logic, 16 FO4s	N/A	676	489	250	116	61.3	24.0	10.4
logic, 4 FO4s	4160	509	336	131	63.9	35.2	16.0	7.02
logic, 0 FO4s	1130	386	265	99.3	48.8	27.3	13.2	5.57
latches	360	120	82.4	31.9	15.0	7.96	3.73	1.66
SRAM	146	48.8	33.7	12.9	6.31	3.43	1.52	0.670
QS	52.3	34.6	26.8	17.2	12.2	9.53	7.19	5.54

Table 3.3: Critical charge for an SRAM cell/latch/logic chain by feature size and pipeline depth

The effect of the declining Q_{CRIT}/Q_S ratio can be directly observed in Figure 3.12, which plots this ratio for a single SRAM cell, pipeline latch, and logic chain by technology generation. This graph shows that Q_{CRIT}/Q_S of SRAMs is relatively small for all feature sizes, confirming that reductions in Q_{CRIT} due to device scaling will have only a secondary effect on SER for SRAM circuits. The Q_{CRIT}/Q_S ratio for latches is significantly larger than SRAMs in the 600nm technology, but decreases to nearly the same level as SRAMs by the 180nm technology generation. Device scaling in memory elements affects the critical charge and charge collection efficiency almost equally because smaller transistors are more sensitive to a particle strike but have very little sensitive volume for charge collection. Logic shows the largest decrease in the Q_{CRIT}/Q_S ratio, but remains above the level of SRAM cells and latches even in the 50nm technology generation.

Effects on latching-window masking: We also performed experiments to determine the effects of technology trends on latching-window masking. We recomputed the SER of combinational logic with the assumption that any charge larger than Q_{CRIT} will result in a soft error. Then we divided this new value for SER by the original SER value to obtain a ratio that indicates the effect of latching window masking for a given technology generation and pipeline depth. Figure 3.13 presents the results of this analysis. The basic trend in these results is that the effect of latching-window masking decreases with decreasing feature size.

While the results for the 600nm technology generation do not follow this trend, this

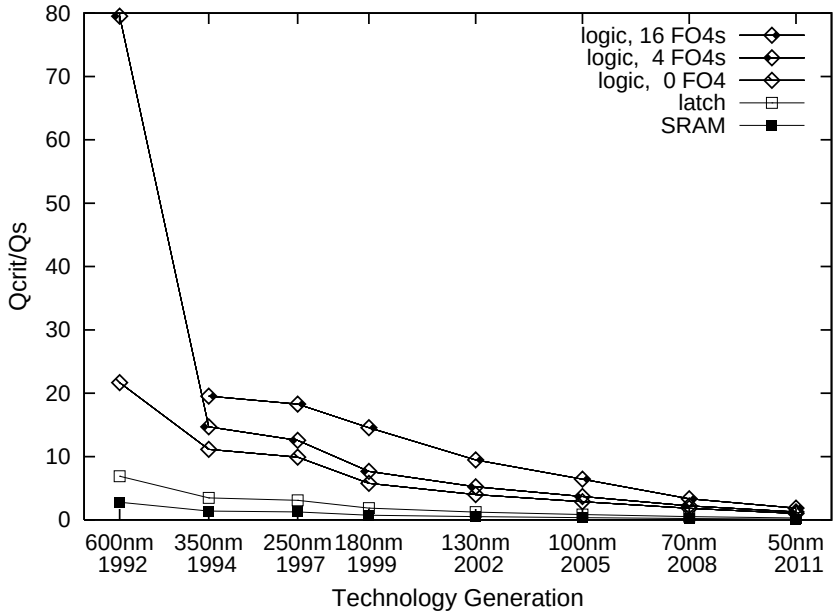

Figure 3.12: Ratio of critical charge to charge collection efficiency for SRAM/latch/logic

is an anomaly caused by our model for electrical masking. The source of this anomaly appears to be the unusually low value of 0.16 for the V_{rise} gate switching voltage for the 600nm technology, which comes from our calibration procedure for the Horowitz gate delay model. We confirmed through separate Spice simulations that the latching-window masking effect for the 600nm technology is significantly higher than indicated in the graph, and in agreement with the trend for the remaining technology generations.

As feature size decreases, latching-window masking decreases because latches have much shorter response times and so have smaller latching windows. This increases the probability that a pulse of a given duration will overlap the window (see Equation 3.8), and hence reducing the effect of latching-window masking. Within a technology generation, the latching-window masking effect decreases with decreasing number of gates between latches. This is because at lower clock rates the latching window occupies a smaller fraction of the clock period. In summary, our results demonstrate that latching window masking is reduced by both reduction in feature size and higher degrees of pipelining.

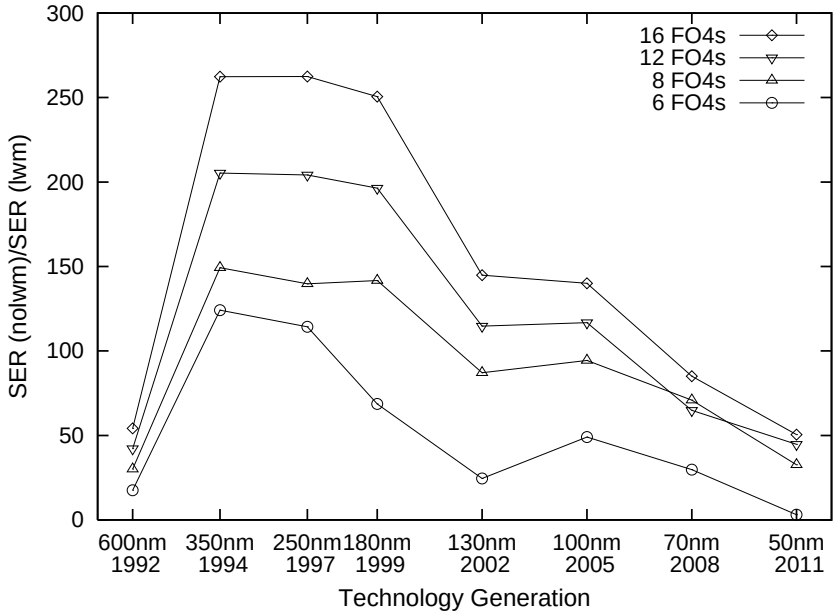

Figure 3.13: Effect of latching-window masking. The y-axis plots the ratio of SER without accounting for latching-window masking and the SER with latching-window masking. The SER without latching-window masking is computed assuming that any charge greater than Q_{CRIT} results in a soft error.

3.4.2 Processor Soft Error Rate

Now we determine how soft errors in SRAM cells, latches, and logic circuits contribute to the SER of the entire processor chip for future microprocessor technologies. As feature sizes decrease, the number of transistors that can be placed on a fixed size die increases quadratically, creating significantly greater opportunity for soft errors. Since the rate of soft errors is different in SRAM cells, latches and logic, the SER of the processor will depend on the chip area devoted to each type of device. To estimate the SER of the entire chip we have developed a chip model that describes the transistor decomposition into logic, SRAMs and latches. From the chip model we determine the total number of SRAM bits, latches and logic chains and then scale the per unit SER of each circuit by their number on the chip to obtain the SER/chip.

Device size	Total	SRAM	Latches	Logic gates
600nm	5.17 M	4.07 M (78.8%)	0.06 M (1.2%)	1.03 M (20.0%)
350nm	15.2 M	11.9 M (78.8%)	0.19 M (1.2%)	3.04 M (20.0%)
250nm	29.7 M	23.4 M (78.8%)	0.37 M (1.3%)	5.95 M (20.0%)
180nm	95.0 M	74.8 M (78.8%)	1.18 M (1.2%)	19.0 M (20.0%)
130nm	229 M	181 M (78.8%)	2.87 M (1.3%)	45.9 M (20.0%)
100nm	440 M	347 M (78.8%)	5.50 M (1.2%)	88.1 M (20.0%)
70nm	919 M	724 M (78.8%)	11.4 M (1.3%)	183 M (20.0%)
50nm	1818 M	1431 M (78.8%)	22.7 M (1.3%)	363 M (20.0%)

Table 3.4: Transistors per chip for 16 FO4 pipeline using quadratic scaling assumption

Pipeline depth	SRAM bits	Latches	Logic gates
16 FO4s	1995 K (78.8%)	32 K (1.2%)	507 K (20.0%)
12 FO4s	1984 K (78.3%)	42 K (1.7%)	507 K (20.0%)
8 FO4s	1963 K (77.5%)	63 K (2.5%)	507 K (20.0%)
6 FO4s	1942 K (76.7%)	84 K (3.3%)	507 K (20.0%)

Table 3.5: Chip model for 350nm device size

Chip Model: We used the Alpha 21264 microprocessor as the basis for constructing our chip model. The Alpha 21264 was designed for a 350nm process and has 15.2 million transistors on the die [47]. Based on a detailed area analysis of die photos of the Alpha 21264 [43], we concluded that approximately 20% of transistors are in logic circuits and the remaining 80% are in storage elements in the form of latches, caches, branch predictors, and other memory structures. Our chip model applies this basic allocation to all feature sizes. The total number of transistors per chip is scaled quadratically from the baseline Alpha 21264 based on feature size. Table 3.4 presents the total number of transistors per chip, and the transistors devoted to each circuit class for each technology based on this assumption.

A typical SRAM bit requires 6 transistors, the level sensitive latch we use in our model consists of 6 transistors, and we assume each logic gate also uses 6 transistors. These assumptions are realistic and using slightly different values for these numbers will not affect the overall trend noticeably.

The allocation of memory element transistors to SRAM cells and latches depends on

the number of latches required by the processor pipeline, which depends on pipeline depth. We allocate one latch for each logic chain, and the remaining memory element transistors are allocated to SRAM cells. Table 3.5 illustrates how our model allocates transistors to SRAM bits, latches, and logic gates in the 350nm feature size for four pipeline depths. Our chip model is summarized in the following equations:

$$
\begin{aligned}
\text{total\_transistors} &= 15.2 \text{ million} \times \left(\frac{350\text{nm}}{\text{feature\_size}}\right)^2 \\
\text{logic\_chains} &= \frac{\text{logic\_transistors}}{\text{gates\_per\_logic\_chain} \times \text{transistors\_per\_gate}} \\
\text{latches} &= \text{logic\_chains} \\
\text{SRAM\_bits} &= ((\text{total\_transistors} \times .80) - (\text{latches} \times 6))/6
\end{aligned}
$$

Results: Using the SER of individual elements shown in the previous section and our chip model, we computed the SER/chip for each class of components for each technology generation and pipeline depth of our study. The results are presented in Figure 3.14. As discussed above, SER/chip of SRAM shows little increase as feature size decreases. To simplify the graph we only plot SRAM data for one pipeline depth. Pipeline depth has no noticeable effect on the SRAM SER/chip, since the percentage of chip area allocated to SRAM changes very little. SER/chip in latches increases only slightly for all pipeline depths, a combined effect of the relatively constant SER/latch and the increasing number of latches at smaller feature sizes. SER/chip of latches increases for deeper pipelines, due solely to the greater number of latches required for deeper pipeline microarchitectures.

SER/chip in combinational logic increases dramatically from 600nm to 50nm, from 10^{-7} to approximately 10^{2}, or nine orders of magnitude. This is simply the composition of a 10^{6} increase in SER per individual logic chain and more than 100 times increase in logic chains per chip. At 50nm with 6 FO4 pipeline, the SER per chip of logic exceeds that of latches, and is within two orders of magnitude of the SER per chip of unprotected

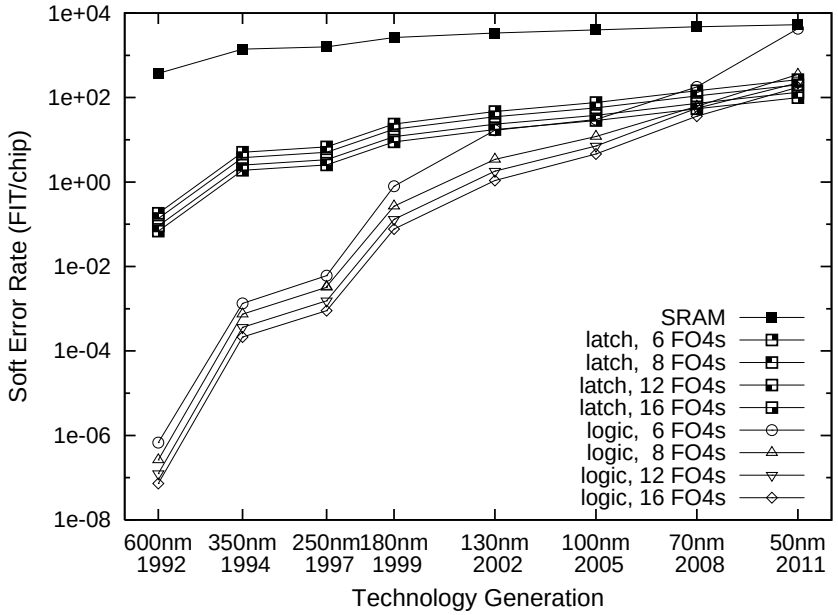

Figure 3.14: SER/chip for SRAM/latches/logic

memory elements. Mainstream microprocessors from Intel [68] and other vendors [43] have employed ECC to reduce SER of SRAM caches at feature sizes of up to 350nm. For processors that use ECC to protect a large portion of the memory elements on the chip, logic will quickly become the dominant source of soft errors.

3.5 Discussion

In this chapter, we have explored the impact of device and pipeline scaling on the soft error rate of processor building blocks: SRAM cells, latches, and combinational logic. The primary focus of our study has been to establish the basic trend in SER of combinational logic and the major influences on this trend. Our model considers the effects of device scaling and superpipelining trends, and the corresponding effects on electrical and latching window masking. This section discusses other factors that influence the SER of these circuits and combinational logic in particular, but are not considered in our model to simplify the model construction and analysis.

System level SER: In this chapter, we have only considered the device and circuit level masking effects that lead to an observed SER that is lower than the raw SER based only on the particle flux. These masking factors can be together referred to as the *electrical masking factor* or the *electrical vulnerability factor*. Analogously, there are several other masking factors at the microarchitectural, architectural, and application level which further reduce the actual SER observed by the user or the application. We did not consider these masking factors in this chapter since our goal was to understand the impact of technology trends on SER. Chapter 6 explores these masking factors in greater detail.

Circuit Implementations: We restricted our analysis to static combinational logic circuits and level-sensitive latches. Modern microprocessors frequently employ a diverse set of circuit styles, including dynamic logic, and latched domino logic, and a variety of latches, including edge-triggered flip flops, with different combinations of performance, power, area, and noise margin characteristics. We believe our model could be extended to include these additional circuit styles and latch designs.

Figure 3.15a illustrates the static 3-input NAND gate used in our model. The transistors are sized so that the worst case rise/fall time of the gate is equal to an inverter with NMOS width 'W' and PMOS width '2W' which makes its drain area larger than that of the equivalent inverter. In this gate there are 3 nodes where a particle can strike and have an effect on the output, but due to capacitive charge sharing only a direct hit to the output node has a significant contribution to soft error rate.

The use of dynamic logic could substantially increase the SER, since each gate has built-in state that can reinforce an error pulse as it travels through a logic chain. Figure 3.15b shows a NAND gate implemented in latched CMOS domino logic. Note that the cell contains a feedback inverter whose purpose is to hold the value of the output constant. Typically this inverter is designed with a low switching voltage to reduce delay through the circuit, lowering its noise margin and making it more susceptible to soft errors.

We use level sensitive latches in our pipeline model because they occupy less area

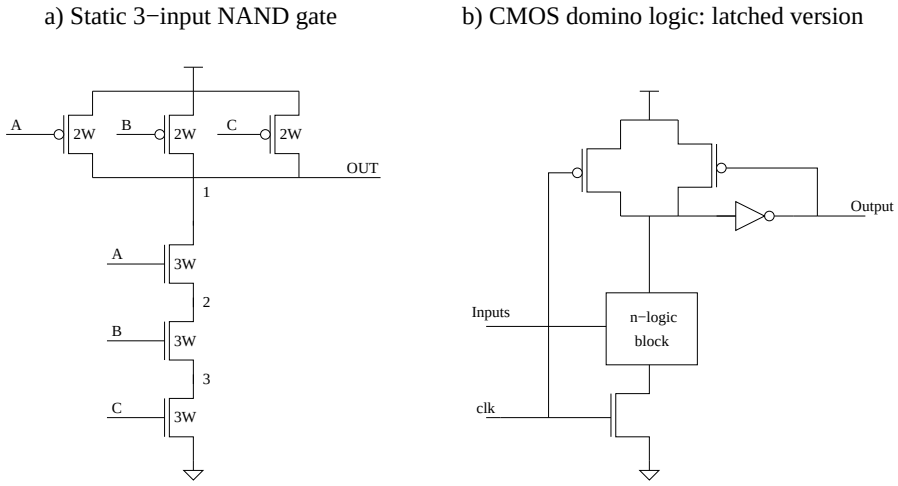

Figure 3.15: Circuit implementations: a) Static 3-input NAND gate, b) CMOS domino logic : latched version

than edge triggered flip-flops and so are more suitable for superpipelining. However, the critical charge for this type of latch is typically smaller than that of a static edge-triggered latch. If we had used an edge-triggered, we expect that the estimated SER for both latches and logic would be reduced.

For superpipelined microarchitectures, latches should be designed to be very fast and occupy minimal area. A common technique for increasing latch speed is to increase the widths of the latch transistors, but this increases the area of the sensitive regions in the latch, thus increasing the potential for soft errors. One approach to reducing latch area is to eliminate the passgate typically placed within the feedback loop of the latch. This also increases the positive feedback in the loop which makes the latch faster and more capable of latching weak input pulses, but increases the likelihood of latching an error pulse caused by a particle strike. These points illustrate the importance of design choices on the overall SER.

Logical Masking: Logical masking is another masking effect that inhibits soft errors in combinational logic and could have a significant effect on the SER. Since our model

places every logic gate on an active path to a latch, we do not account for the the effect of logical masking. Incorporating logical masking would increase the complexity of the model significantly, since the model would need to consider actual circuits and associated inputs. Massengill *et al.* developed a specialized VHDL simulator that could analyze soft faults in an actual circuit and model the effects of logical masking [69]. They found that effect of logical masking on SER depends heavily on circuit inputs. More recently, both Zhang et al. [70], and Constantinides et al. [71] show that failure to model logical masking can over-estimate SER by over an order of magnitude.

Effects similar to logical masking can also occur in memory elements. For example, if a soft error occurs in a memory element that holds dead data – data that will not be used again – it is in some sense logically masked. Another example is a soft error in a memory structure such as a branch predictor, which may lead to reduced performance but not produce incorrect results. Due to the difficulty in modeling these effects, we have chosen to exclude all forms of logical masking in memory elements or logic from this model. In Chapter 6, we extend this analysis to take in into account such microarchitectural masking factors that reduce the vulnerability, and improve the accuracy of processor SER estimation.

Alpha Particles: Our study only considers soft errors resulting from high-energy neutrons. Another important source of soft errors in microprocessors is alpha particles that originate from radioactive decay of uranium or thorium impurities in chip and packaging materials. In sub-0.25um technologies with decreasing supply voltage and node capacitances, the SER due to alpha particles presents a major reliability concern to logic processes because of the quadratically decreasing critical charge [72–74]. Packaging alternatives such as lid coat or flip chip strongly influence the soft error rate induced by alpha particles. Alpha particle SER increases more rapidly with decreasing critical charge than neutron induced SER [74–76]. For circuits with Q_{CRIT} in the range of 10-40 fC, the alpha particle SER becomes comparable to that of neutron SER [8]. In our experiments, this range corresponds to SRAM cells and latches in 180nm and later technologies and logic circuits in 50nm and

later technologies. Our model could be adapted to estimate the SER due to alpha particle radiation. This would require a technology-scaled alpha particle model for the charge collection efficiency and the time constant for the NMOS and PMOS transistors. A key input to this model would be the expected flux of alpha particles, which is determined mainly by package design.

Fabrication Technology: The process used to manufacture a semiconductor device has a significant effect on the SER of the device. New process technologies such as silicon-on-insulator (SOI) have been shown to significantly reduce SER [8, 9]. The SOI process embeds an oxide (insulator) layer in the substrate which significantly decreases the volume available for charge collection. This reduces the amount of charge collected due to a particle strike, which decreases the probability of soft errors. However, the floating body of the device can charge up to considerable voltages leading to a reduction in effective threshold voltage, making the circuit more susceptible to noise. Substantial design effort is required to avoid the negative effects of the changing body voltage.

In this study, we used a model proposed by Hazucha et al. (Equation 3.1) to project the SER of combinational logic at future technologies. One of the critical components is the scaling model used for Q_S captured in Equations 3.9 and 3.10. Since the model in Equation 3.1 has an exponential dependence on Q_S, deviations in Q_S from the scaling model, due to differences in device parameters, can make dramatic difference in the SER at future technologies.

3.6 Related Work

Although ours was the first study to model the effect of both technology scaling and super-pipelining on the soft error rate of combinational logic [67], previous experimental work had been done to estimate the soft error rate of storage and combinational logic in existing technologies [53, 58, 59, 77, 78].

Soft error models: While we used the model proposed by Hazucha et al. [67], another method for estimating the neutron-induced SER uses the Modified Burst Generation Rate model [79]. This method uses nuclear theory to calculate the collected charge resulting from a particle strike. IBM developed the SEMM (Soft-Error Monte Carlo Modeling) program to determine whether chip designs meet SER specifications [80]. The program calculates the SER of semiconductor chips due to ionizing radiation based on detailed layout, process information and circuit (Q_{CRIT}) values.

Prior work in SER estimation of logic circuits: Prior to this study, some research had also been done to estimate the SER in combinational logic. Liden *et al.* compared the soft error rate due to direct particle strikes in latches with the soft error rate from error pulses propagating through the logic gates [53]. They considered a circuit implemented in 1000nm technology clocked at 5MHz. They conclude that the errors are predominantly due to direct strikes to latches and only 2% of the total observed errors are from the logic chain. We have shown how technology trends will lead to a significant increase in the SER at low feature sizes and high clock rates. Baze *et al.* studied electrical masking in a chain of inverters and concluded that for pulses that successfully get latched electrical masking does not have any significant effect on SER [81]. They also allude to various parameters such as the chip model and the clock rate as factors that might affect the impact of this effect on the overall SER. Our results show that electrical masking does have a significant effect on the SER, and this effect is not diminishing with decreased feature size. Buchner *et al.* investigated latching window masking in combinational and sequential logic [82]. They concluded that while the SER of sequential logic is independent of frequency, combinational logic SER increases linearly with clock rate. Our results confirm that the trend of increasing clock rate due to increased processor pipelining increases the SER of logic circuits.

Improving the speed and accuracy of logic SER estimation: Our study triggered a lot of research in this area, specifically focused on increasing the accuracy of the different mod-

els used to estimate the SER, and increasing the speed of SER estimation without sacrificing accuracy. Seifert et al. demonstrate that latching-window masking is a strong function of the propagation delay of combinational logic, and can be significantly less than 50% for high frequency latch designs [83]. Wang et al. proposed a table lookup based scheme to accurately model the non-linear behavior of submicron devices, and achieve almost an order of magnitude increase in the precision of electrical masking [84]. Constantinides et al. analyzed both logical masking and latching window masking accurately and showed that they account for more than 50% of the masked errors [71]. Zhang et al. used circuit simulation to accurately account for electrical masking under different operating conditions such as supply voltage, clock frequency, latch design, setup and hold time, gate sizing, logic depth, and fan-out. They used a combination of logic simulation, and probability and graph theory to account for logical masking. Using this hybrid methodology they compute combinational logic SER within 4% accuracy and with a million-fold speedup [85]. Zhang et al. proposed FASER, a fast static analysis technique for estimating combinational logic SER, that achieves over 90,000X speed-up [70]. The efficiency is achieved by using binary decision diagrams for propagating error pulses, and the accuracy is maintained by using Spice based cell library characterization. Rao et al. also propose an efficient algorithm for computing combinational logic SER, accounting for all the three masking factors, and achieve linear runtime with the size of the circuit [86]. A good overview of several other techniques proposed for estimating logic SER is provided in [85].

Latch SER scaling: Several recent studies have focused on analyzing the scaling trends of latches at future technologies. Seifert *et al.* used experiments and simulation to determine the trend of soft error rate in the family of Alpha processors [87]. They conclude that the alpha particle susceptibility of both latches and memory circuits has decreased over the last few process generations. Karnik et al. conservatively project SER-per-latch to either increase by 8% every generation if supply voltage scales linearly (0.7x) across technologies, or remain constant if voltage scaling slows down, and supply voltage decreases only by 0.8x

every generation [88]. However, they also conclude that even with slowing voltage scaling the SER-per-latch can increase by up to 20% every generation depending on the device and charge collection parameters. More recently, Seifert et al. concluded that the SER-per-latch is like to remain constant across technologies [74]. These results are consistent with the results in this dissertation which show a relatively constant SER-per-latch across technologies.

Combination logic SER scaling: With the continued scaling of feature sizes, soft error rate in combinational logic is becoming a growing concern. The ITRS roadmap recommends cost-effective soft error derating, detection, and correction schemes for combinational logic circuits at multiple levels of the design all the way from the device to the architectural level [4].

Zhang et al. corroborate our study by showing that SER of combinational circuits are comparable to or exceed that of SRAMs of similar area, but differ in the rate of increase with decreasing feature size since they model logical masking [85]. They also make the key observation that combinational logic SER is very dependent on the width of the latching window. Seifert et al. further strengthen this conclusion by showing that soft errors in combinational logic account for the dominant fraction of the total SER of an 8 bit adder at 90nm [74]. Further, they demonstrate that α-induced combinational logic SER increases by over five orders of magnitude with decreasing Q_{CRIT}, in agreement with our neutron SER scaling trends for combinational logic. Blome et al. analyze the masking and propagation behavior of faults in combinational logic, and identify reasons why they are more dangerous than faults in sequential elements [89]. Most importantly, they show that combinational logic produces a larger average number of manifested errors per fault due to the significantly higher probability of multi-bit errors from logic fan-out. These results corroborate our study, and emphasize the need for new reliability mechanisms that extend SER protection to combinational logic and latches.

However, in contrast to our projections, Seifert et al. measure a slight decrease

in the neutron-induced combinational logic SER of specific circuits from 90nm to 65nm. Further, they report a constant SER per diffusion area across the range of observed Q_{CRIT} values, and ascribe this to the slowdown in voltage and frequency scaling, and the reduction in the charge collection efficiency. On the other hand, the SER per diffusion area in our model is exponentially dependent on $-Q_{CRIT}/Q_S$, which is projected to uniformly increase across technologies (see Figure 3.12). We believe that these differences are primarily due to the modeling infrastructure. While we use analytical models and ignore logical masking, they use real SPICE simulations to compute Q_{CRIT}, and thus naturally account for electrical, logical, and latching-window masking. Further, while we use the model proposed by Hazucha et al. with an exponential dependence on $-Q_{CRIT}/Q_S$, they use experimentally collected failure data to accurately calibrate their SER model and its dependence on Q_{CRIT}, Q_S, and the diffusion area. We believe that more investigation is required to accurately identify the reasons behind the differences.

3.7 Summary

We have presented an analysis of how two key trends in microprocessor technology, device scaling and superpipelining, will affect the susceptibility of microprocessor circuits to soft errors. The primary impact of device scaling is that the on-currents of devices decrease and circuit delay decreases. As a result, particles of lower energy, which are far more plentiful, can generate sufficient charge to cause a soft error. Using a combination of simulations and analytical models, we demonstrated that this results in a much higher SER in microprocessor logic circuits as feature size decreases. We also demonstrate that higher clock rates used in superpipelined designs lead to an increase in the SER of logic circuits in all technology generations.

The primary cause of the significant increase in the SER of logic circuits is the reduction in critical charge of logic circuits with decreased feature size. Our analysis also illustrates the effect of technology trends on electrical and latching-window masking, which

provide combinational logic with a form of natural protection against soft errors. We found that electrical masking has a significant effect on the SER of logic circuits in all technology generations, and this effect is not diminishing with feature size. The effect of latching-window masking is important, but is reduced by both decreasing feature size and increased clock rate of future technology generations. We conclude that current technology trends will lead to a substantially greater increase in the soft error rate in combinational logic than in storage elements.

The implication of this result is that further research is required into methods for protecting combinational logic from soft errors. Recently, a number of schemes have been proposed to detect or recover from transient errors in processor computations. All these techniques are either based on space redundancy or time redundancy [17, 90–92] and have different performance-reliability tradeoffs. In the next chapter, we will discuss the salient features of future distributed architectures that provide the foundation for building low overhead mechanisms for soft error reliability. Building on these features, we will discuss both hardware and software based soft error reliability enhancement techniques in chapter 7.

Chapter 4

Reliability of Distributed Architectures

Conventional out-of-order superscalar processors face several challenges in achieving technology scalable performance including but not limited to accessing large centralized structures, associative wakeup in large instruction windows, high bandwidth register renaming, full operand bypass networks that scale poorly with increasing issue width, and per instruction access to a centralized register file [34]. These challenges also push the future technology limits of power consumption and design complexity, and have motivated architects to pursue more technology scalable designs involving extensive partitioning of the microarchitecture structures, and implementing the processor functions in a distributed fashion on this partitioned execution substrate to efficiently extract parallelism from the application. There have been several distributed architectures that have been proposed to achieve technology scalable performance [47, 93–100]. In this Chapter, we identify the underlying common principles of distributed architectures that provide technology scalability, and argue that these features provide the foundation for implementing low overhead mechanisms for improving hard and soft error reliability.

The rest of the chapter is organized as follows. Section 4.1 identifies the com-

mon principles of distributed architectures. Choosing a specific architecture is necessary for quantitative evaluation, and Section 4.2 describes in detail the distributed TRIPS architecture we use to implement and evaluate our mechanisms. Section 4.3 argues how the principles of distributed architectures are synergistic with the approaches for improving reliability. Finally, Section 4.4 summarizes our discussion, and provides a good starting point for evaluating reliability enhancement techniques that exploit these synergies.

4.1 Distributed Architecture Principles

We identify three important principles for a distributed architecture:

Modular design: Employ resource partitioning at different granularities to eliminate access to large centralized resources. The microarchitecture is composed by connecting the different modules as necessary.

Cooperative hardware / software techniques: Rebalance the hardware and software effort to efficiently mine parallelism from the application, and exploit the distributed substrate to achieve high performance.

On-chip networks: Utilize well-defined, scalable networks for communication between the different modules, and employ explicit communication protocols on these networks for managing global latency.

Together the three design principles enable the architecture to achieve technology scalable performance. Several distributed architectures have been proposed based on these principles, but each employing different mechanisms, and making different tradeoffs based on the specific application and technology constraints [47, 93, 96–100]. Resource partitioning can be restricted to a few structures, a subset of the pipeline stages within a uniprocessor, or can be extended to the chip level in the form of multiple independent processing units

on a single chip. While on-chip networks have been proposed to interconnect memory components for supporting cache coherency, it can be expanded to enable direct communication between the distributed processing components with the help of dedicated networks for distributing instructions, commands, and operand data. Finally, there have been several approaches explored in industry and research for dividing the responsibility of finding parallelism in the application, and the task of distributing the parallel units of computation to the processing units, between the hardware and software. We expand on each of the three principles below with three specific distributed architecture design styles as examples.

4.1.1 Modular Design

Modularity in the design can be applied at different granularities. Clustering is a modular design methodology that can be applied within a processor. The Alpha 21264 was the first commercial superscalar processor to adopt a clustered design [47]. The execution stage of the integer pipeline was partitioned into two clusters, with each cluster containing two integer units. The floating-point functional units are placed in a cluster of their own. Clustering provides many advantages to a design. By partitioning the design, each cluster requires smaller structures and simpler instruction scheduling logic compared to a non-clustered design, while supporting the same aggregate throughput across all the clusters. Since the smaller structures have lower access latency, a clustered design can run at higher frequencies.

The Alpha 21264 design partitions only a subset of the pipeline stages, register read, issue, and execute, into two clusters. Register renaming and instruction scheduling are still centralized and rely on conventional monolithic structures. Many modern implementations of superscalar processors employ this clustering topology and implement a separate integer and floating point cluster. Since clustering is only applied to a subset of the pipeline stages it does not completely solve the technology scalability problem. Other designs for clustered superscalar processors have been proposed in research to address this problem [93].

More recently, the microprocessor industry has migrated toward integrating multiple moderately complex processors on a single chip to achieve a balance between the single thread performance offered by each core and the multi-threaded performance across all the cores [101–105]. The extra concurrency provided by the multiple processors can be traded for the clock frequency of each processor to keep the power consumption of the chip within acceptable limits. Further, each processor in the chip multiprocessor (CMP) also provides a natural granularity for independently implementing dynamic power management techniques such as DVS. Current mainstream designs use separate integer and floating point clusters within a processor, and CMP based design at the chip level to improve overall scalability.

In conventional CMP designs the granularity of each processor core on the chip, and hence the number of processors on the chip, are fixed at design time. Recently, researchers have proposed tiled architectures which use a small set of unique processing units or tiles to compose the overall microarchitecture. The tile-based design methodology allows easier extensibility of the architecture from generation to generation by changing the number or configuration of the different tiles [106]. Further, recent research has demonstrated that a tile-based methodology also provides the capability to dynamically aggregate different number of tiles to form multiple processors with varying configurations [107]. The RAW and CLP design both use an array of simple processors as tiles in the microarchitecture [100, 107]. The TRIPS architecture uses five unique tiles to build the processor core, and eleven unique tiles in all to compose the entire chip [108].

4.1.2 Cooperative Hardware/Software Techniques

The benefit of a clustered design depends on the rate of inter-cluster dependences and transfers, since each communication operation incurs extra cost. The assignment of instructions to clusters determines the number of inter-cluster transfers required. Using the compiler to perform cluster assignment is favorable from a technology scalability viewpoint, but can be

sub-optimal because of the lack of dynamic knowledge at the compiler. Hardware based dynamic cluster assignment has the potential for higher performance, but introduces another scalability bottleneck. The cost of each communication operation depends on whether it is implemented by dynamically inserting operand packets into the inter-cluster network, or by adding explicit cluster transfer or *move* instructions to the program. While compiler-inserted *move* instructions occupy extra space in the instruction window and compete with normal instructions to increase issue contention, directly inserting packets into the network can potentially extend the critical path and impact clock frequency.

In the context of a CMP, the programmer, runtime system, or the compiler is responsible for parallelizing a program into multiple threads which can then be executed concurrently on the multiple processors [109]. One such example is the Multiscalar processor proposed by Sohi et al. [97]. In the Multiscalar execution model the program's control flow graph (CFG) is partitioned into tasks and each task is assigned to a processor. Multiple tasks execute speculatively in parallel on different processors, and have to be appropriately flushed if a mis-speculation is detected. Inter-task register dependences are marked explicitly by the compiler and delivered using specialized networks.

Since the number of processors and configuration of each processor is fixed at design time in a CMP, it results in suboptimal operation as the number and type of available threads change over time. Architectures such as RAW and CLP solve this problem by providing support for mapping instructions from a single program to a subset of the tiles based on the program requirement. The hardware and software cooperate to provide the capability for dynamically aggregating multiple tiles together based on the computation, and memory requirements of the program. RAW uses static compiler-controlled mapping of instructions to processors, and statically scheduled execution and routing of values between them, thus greatly reducing the burden on the hardware, while being limited by the static knowledge available at the compiler [100]. On the other hand, the CLP design uses a hybrid execution model that uses static mapping of instructions to tiles, but dynamic scheduling of execution

and communication.

4.1.3 On-chip Networks

Distributed architectures implement explicit mechanisms to manage the latency between the different modules. Communicating values from the local cluster to the register file of the *remote* cluster requires an additional clock cycle in the Alpha 21264, and is transmitted on a dedicated inter-cluster network [47]. Each processor in a CMP can run an independent program, and communication between the applications, if required, is performed through memory using the standard cache coherence network. The processing units or tiles in a tiled architecture are simple, and are inter-connected by a dedicated scalar operand network (SON) that enables fast inter-tile communication to achieve high performance [100, 110]. Direct communication between the tiles is faster and more efficient than communicating only through memory as is done in conventional CMP designs.

4.1.4 Summary

The three different distributed architectures discussed use the same three basic principles to build different mechanisms:

- Modularity at different granularities
  - Cluster
  - Processor
  - Tile
- Software cooperation
  - Assignment of instructions to clusters, and inter-cluster transfers
  - Extracting threads for execution on a CMP, and assignment of threads to processors

  - Assignment of instructions to tiles, composing processing units/tiles to form a logical processor

- On-chip networks for communication

  - Inter-cluster network
  - Cache coherence network
  - Scalar operand network

Future distributed architecture will increasingly embrace these three principles to achieve technology scalable performance, power consumption, and design complexity. In this dissertation, we use the TRIPS architecture proposed by Sankaralingam et al. to implement and evaluate our mechanisms [111].

4.2 The TRIPS Architecture

This section describes the features of the TRIPS architecture in detail. We then discuss how these distributed features can be exploited to improve reliability at low overhead.

4.2.1 Modular Design

The TRIPS architecture uses a tile-based modular design. Unlike conventional architectures, all major processor structures are partitioned into tiles, thus eliminating access to centralized structures and reducing the pressure on cycle time. The TRIPS architecture uses each type of tile multiple times as necessary, striking a balance between the benefits of partitioning and the distribution overhead, to compose the overall microarchitecture. The processor core is built out of five unique tiles each responsible for a different function: 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), four instruction tiles (IT), and one global control tile (GT).

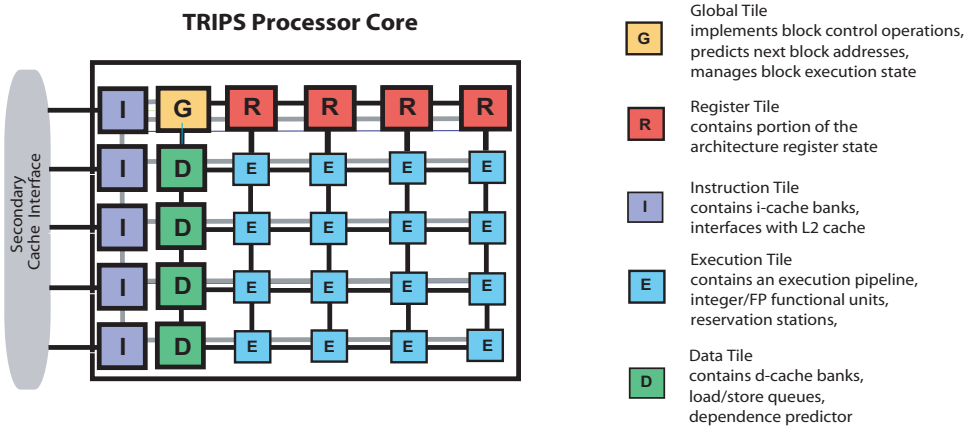

Figure 4.1: TRIPS processor microarchitecture

Figure 4.1 illustrates the organization of the TRIPS core. The execution resources are distributed across 16 ETs, which are organized as a two-dimensional array. Each ET contains an integer unit, a floating point unit, an operand router, and instruction and operand buffers or reservation stations for storing instructions and their operands. Each ET contains 64 instruction buffer entries. A 4×4 grid, with 64 instruction buffer entries at each ET, thus represents an instruction window of 1024 instructions. The architectural register file is banked and distributed across four RTs placed on the top of the ETs. The RTs also contain *read* and *write* queues that buffer register inputs, and un-committed register outputs and implement register forwarding. The level-1 instruction and data caches are also banked and distributed across the four ITs and DTs respectively, and placed along the left of the execution array. The DTs also contain the load-store queue (LSQ) and associated logic necessary for load-store ordering, and the miss-status-handling-registers (MSHRs) for miss handling.

The execution resources can be naturally partitioned for mapping up to four threads when operating the TRIPS processor in multi-threaded mode. While similar to simultaneous multithreading [112] in that the execution resource and memory banks are shared, the multi-threaded mode statically partitions most major processor queues including but not

limited to the instruction and operand buffers in the ETs, the read and write queues in the RTs, and the LSQ in the DTs, across the four threads. No additional space is required in the processor queues, since the same storage used to hold speculative state in single-threaded mode can instead store state from the multiple threads. For instance, while a program can use all the 64 instruction buffer entries in single-threaded mode, each thread is restricted to use only 16 buffer entries in multi-threaded mode. Specifically, thread 0 uses the first 16 buffer entries, thread 1 uses the next 16 buffer entries and so on. Therefore, across all the ETs each thread has a instruction window that can hold 256 (16x16) instructions. Four copies of the architectural register file are used to accommodate the registers of up to four threads.

4.2.2 Cooperative Hardware/Software Techniques

The TRIPS architecture uses three main techniques to build an innovative execution model for technology scalability.

Block-atomic execution model: TRIPS implements a block-atomic execution model, in which a block of instructions is treated as a single unit as it flows throw the logical pipeline to perform fetch, decode, register read, execute, and commit. Hence, the block-atomic execution model amortizes the per-instruction logic and bookkeeping overheads incurred in conventional instruction-atomic architecture such as current day superscalar processors. Further, the block-atomic execution model also reduces the number of branch predictions, register file accesses, and the complexity of the register renaming hardware. The TRIPS compiler converts programs into blocks of instructions, which are similar to hyperblocks [113]. TRIPS blocks have a single entry point, no internal control transfers, and possibly multiple exit points. Each block has a static set of state inputs but potentially a variable number of state outputs, depending on the specific exit point from the block. Each block can contain up to 128 instructions.

Static Instruction Placement: Superscalar architectures dynamically associate an instruction with a specific reservation station entry, which increasingly becomes a scaling bottleneck with larger window sizes [46, 93]. The TRIPS architectures solves this problem by exposing the distributed instruction window in the ETs to the compiler which performs static instruction placement. Each TRIPS block is scheduled as a whole on to the distributed instruction window in the ETs [114, 115]. The compiler statically assigns each instruction in the TRIPS block to a specific ET instruction buffer entry, and explicitly encodes inter-instruction dependences within the block. The 64 instruction buffer entries in each ET are divided into 8 block partitions, each containing 8 entries. The compiler maps the 128 instructions in a block across all the sixteen ETs, with one block partition in each ET containing eight instructions. Since the instruction buffer in each ET has eight block partitions, eight TRIPS blocks can be simultaneously mapped on the processor to support speculative execution. All the major processor queues including the operand buffers in the ETs, the read and write queues in the RTs, and the LSQ in the DTs are also statically block-partitioned in a similar fashion to support up to eight speculative blocks.

Direct Instruction Communication: The third important innovation in the execution model improves the capability to execute dependent instructions efficiently. Since prior research has shown that using associative matches in the instruction window to wake-up dependent instructions is not scalable [46], TRIPS adds support in the ISA to allow dependent instructions to directly communicate with each other, completely eliminating the need for associative matches. Unlike a conventional architecture which includes with an instruction the destination register specifier that is used for performing the associative look-up, the TRIPS ISA instead explicitly encodes in each instruction the location of the instructions which need its result. The location information includes both the specific ET, and the instruction buffer entry in it. Using this explicit instruction target encoding allows the architecture to replace the associative look-up, with a direct mapped look-up of the instruction buffer at the target ET. While block primary register inputs and outputs must be fetched

from and committed to the architectural register file, block temporaries, which are values that are produced and consumed entirely within the block, can be forwarded directly from producers to consumers using this target information. Further, the on-chip networks discussed in Section 4.2.3 provide an efficient physical substrate for performing this direct instruction-to-instruction communication of block temporaries.

Base Scheduler Algorithm

An example schedule of instructions on a 2×2 execution array is shown in Figure 4.2. `Read` instructions are used to fetch values from the register file to the consumer instructions. Block register outputs are produced by `write` instructions. In the TRIPS ISA, instructions do not encode their source operands like in a conventional RISC ISA, instead they explicitly encode the locations of their children. Figure 4.2 shows the instruction encoding for the above example. For example, the `add` instruction placed at location [0,1,0], upon execution, forwards its result to the `LSH` instruction placed at location [1,1,0]. The explicit concurrency expressed in the ISA, and static mapping of instructions to resources naturally allows for a scalable and modular microarchitecture implementation. Furthermore, the physical instruction layout corresponds to the dataflow graph, and the physical layout of ALUs is exposed to the instruction scheduler, so that the wire and communication delays can be used to help the scheduler minimize the critical path.

TRIPS uses an algorithm called *spatial path scheduling* (SPS) that explicitly reasons about program paths and the physical paths they get mapped onto, when mapping a block to the execution array [115]. Specifically, the goal of the scheduler is to find a placement for each of the 128 instructions in a TRIPS block in the distributed (4x4) execution array, such that the completion time of the block is minimized. The SPS algorithm accomplishes this objective by using sophisticated static estimates for the inherent instruction level parallelism in the block, static execution and routing latencies between dependent instructions, dynamic latencies due to execution unit and network link contention, and global inter-block critical

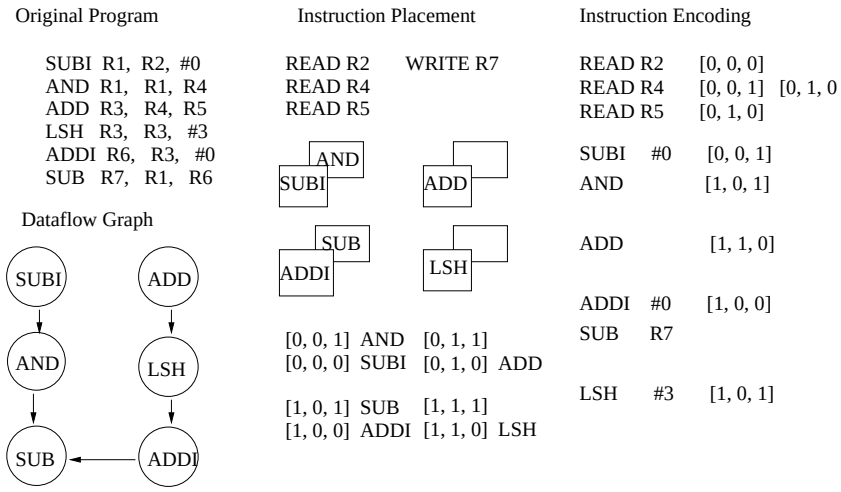

Figure 4.2: Instruction physical layout

paths. The SPS algorithm basically uses *anchor points* and computes the cost of routing operands to and from known anchor points. Anchor points are instructions whose placement is constrained because of reasons that include accessing a known spatial location such as a specific register bank for a read or write instruction, a particular data cache bank for a load or store, the global control tile for a branch instruction, or simply a specific execution tile for an instruction that has already been placed.

For each instruction *i* in the open list, and for each available and legal physical location *slot*, the SPS algorithm computes a *placement cost* associated with placing *i* at *slot*. For each instruction *i* it tracks the *slot* that minimizes the placement cost, and schedules the instruction from the open list that has the largest minimum placement cost. The algorithm adds the unscheduled children of this instruction to the open list and continues until all instructions have been scheduled. The placement cost is the estimated latency of the longest path in the TRIPS block that passes through this instruction. It is computed as:

$$pCost(i, slot) = input\_Latency + util\_Penalty + exec\_Latency + output\_Latency$$

$Input\_Latency$ computes the sum of the arrival time of the critical register input to the block with respect to this instruction, and the static execution and routing delays from that register input to this instruction. The algorithm searches for the register input that is at the start of the longest path containing this instruction through the block, taking into account inter-block register dependences, in computing the critical register input. $Util\_Penalty$ adds the dynamic latencies due to both execution unit and network link contention, and $Exec\_Latency$ accounts for the static execution latency of this instruction. Finally, similar to $input\_Latency$, $output\_Latency$ calculates both the static execution and routing delays to the most critical block register output taking into account global critical paths due to inter-block register dependences. A detailed explanation of the SPS scheduler is provided in [115].

4.2.3 On-chip Networks

The tiles in the TRIPS architecture are interconnected using point-to-point, multi-hop, dynamically routed networks. The architecture uses multiple different dedicated networks for communicating instructions, data, and commands, each implementing protocols that are best suited for the specific purpose. Since each hop in the network takes one cycle, the communication protocols look for the shortest path to minimize latency, and use pipelining to improve network throughput.

Table 4.1 lists the different networks used in the TRIPS architecture. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. Figure 4.3 illustrates how a subset of these networks are used to connect the tiles in the TRIPS processor. The GDN is a simple point-to-point, multi-hop network used for transferring instructions from the ITs to the reservation stations in the RTs and the ETs. Since the tiles are always expected to have enough storage for the incoming instructions, this network does not experience any contention and does not implement any flow control. The GSN is used for signalling block execution status such as i-cache refills, block com-

Micronetwork	Function
Operand network (OPN)	Pass data operands between tiles
Global dispatch network (GDN)	Dispatch instructions to tiles
Global control network (GCN)	Commit and flush blocks
Global status network (GSN)	Transmit information about block completion
Global refill network (GRN)	I-cache miss refills
Data status network (DSN)	Communicate store completion status among the L1 data cache tiles
External store network (ESN)	Determine the completion status of stores in the L2 cache or memory.

Table 4.1: TRIPS processor micronetworks

pletion, and successful retirement of block outputs to architectural state, and also does not implement any flow control. The GCN is a command network and is used to deliver block commit and flush commands from the GT to all the other tiles. The OPN is a 2D mesh network that connects the ETs, RTs, and DTs and is used for communicating operand values. The OPN uses y-x dimension-order routing to transmit scalar operands among the tiles. When routing a packet from a parent to a child tile, the packet first travels in the y-direction (along the column) until it reaches the row of the child, then it travels in the x-direction until it reaches the child tile. The OPN is dynamically routed and experiences contention since multiple OPN packets can be travelling simultaneously from different source tiles to target tiles. The DTs use the *data status network* (DSN) to share information about completion of block stores, and the ESN is used to determine store completion in the L2 and beyond.

4.2.4 Summary

In summary, while TRIPS uses a dataflow execution model within a block, it uses a conventional speculative execution model across blocks. The dependences between block outputs of one block and the block inputs of its successor, along with load-store communication pairs, create the dataflow arcs for the entire program. The output of control transfer instructions which specify the address of the succeeding block are also treated as block

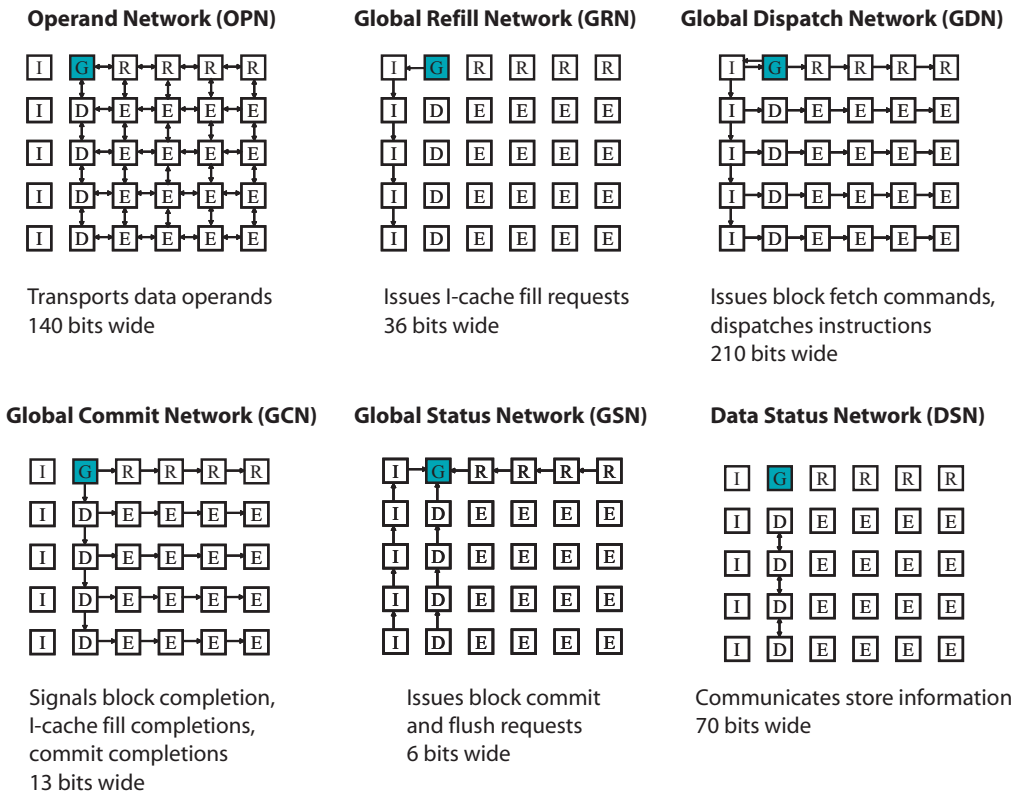

Figure 4.3: TRIPS micronetworks

outputs. The hardware uses control speculation techniques to determine the sequence of blocks and determines the data dependences between blocks through register renaming and load/store dependence checking. The processor uses a "static-placement, dynamic-issue" (SPDI) model in which an instruction is assigned a ET statically, but issued by each ET dynamically depending on the availability of resources and input operands [114]. Section 4.2.5 describes the functions of the different tiles and the networks in the different stages of program execution.

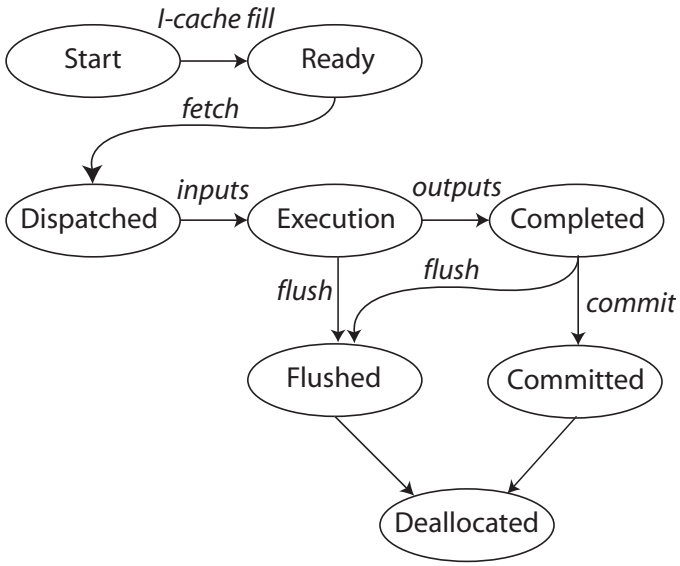

Figure 4.4: Events in the execution of a TRIPS block

4.2.5 Events in the Execution of a TRIPS Block

This section describes the important events in the lifetime of a block execution shown in Figure 4.4. Similar to pipelined execution of instructions, fetch, map, and execution of multiple blocks can be overlapped with the execution of the current block.

Block Allocation: Before the instructions of a block can be fetched, it must be assigned resources in the processor. Resources have to be allocated in multiple tiles including the RTs, ITs, DTs, and the ETs. Each of these tiles provide eight block partitions that are managed in a sequential and cyclical manner. An empty block partition has to be allocated in the RTs for holding the `read` and `write` instructions, in the ITs for holding information regarding instruction cache fetch status, in the Load Store Queue (LSQ) in the DTs for holding the load and store instructions, and in the ETs for holding the normal block instructions. Block allocation has to stall until an empty block partition can be found.

Block Fetch: Once the block instructions have been fetched either from the I-cache or the L2, the hardware initiates **block fetch**. The GT orchestrates the delivery of block instructions, read, and write instructions from the instruction caches to the execution and register tiles using the *instruction dispatch network* (GDN) (see Figure 4.3). The TRIPS architecture supports block-level control speculation to achieve high performance, and uses a next block predictor to speculatively select the next block that should be fetched and executed, while the current block is still executing.

Block Execute: Register read instructions fetch the block register input values from the register file if the value is already resident there.

If the required value is produced by an older block which has not committed its output to the register file, the register renaming logic in the RTs detect this dependence and route the value from the appropriate location in the *Write Queue* corresponding to the older block, to the location in the *Read Queue* corresponding to the current block, from where it is delivered to the consumer instruction at one of the ETs. An instruction in one of the ETs can execute once it has received all its operands. Both the RTs and the ETs use the *operand network* (OPN) to transfer operands. Address computations for load and store instructions execute at the ET and then transmit the addresses, and data values for stores, to the DTs. Each load and store instruction is assigned a *load store identifier* (LSID) to assist the memory system in maintaining program order between the load and store instructions. The DTs send the loaded values back into the execution array using the OPN. The block instructions are held in the reservation stations until the block has completed execution by producing all its register, store, and branch outputs.

Block Completion: A block is said to have completed execution when it has produced all its register and store outputs, and the branch output that points to the next block that should be executed. The DTs use the *data status network* (DSN) to share information about the completed block stores. Both the RTs and the DTs use the *global status network* to

communicate to the GT that all the block register and store outputs have been received. The branch output is directly delivered to the GT using the OPN. The branch, register, and store status information is used to detect block completion.

Block Commit: If the current oldest block has reached block completed state, the block can be *committed*. The GTs use the *global control network* (GCN) to signal to all the DTs, RTs, and the ETs that they can now commit the information to permanent architectural state. Block outputs are written back to the architectural register file and updates to memory are carried out. The GSN is again used to acknowledge to the GT that the updates were completed successfully. Subsequently, a new block can be allocated in the block partition that is freed by this block. In the event of an exception raised by any instruction in the block, the entire block is re-executed after the the exception is serviced.

Block Flush: In the event of a misprediction, the GT uses the GCN to send the flush mask, indicating all the blocks that have to be flushed, to all the DTs, RTs, and ETs. The flush mask is used by each tile locally to flush the state corresponding to all the blocks marked in the flush mask.

Table 4.2 shows the detailed configuration of the baseline TRIPS architecture we use in the rest of this dissertation for evaluating our mechanisms.

4.3 Synergy between Distributed Architecture Principles and Reliability

In Chapter 1, we broadly discussed the different approaches for improving the reliability of a system at different levels of the design. In this section, we return to the same discussion but within the specific context of distributed architectures. In particular, we propose to precisely identify the synergies between the underlying principles and distributed mechanisms in the TRIPS architecture, and the approaches for improving reliability. The rest of

Tile	Composition
GT	Block management state supporting eight blocks in single-threaded mode and two blocks for each of 4 SMT threads in multi-threaded mode, L1 I-cache tags, 128 TRIPS blocks, 2-way set-associative, 16-entry, fully-associative instruction TLB, 84 Kbit next-block predictor that includes a local/global adaptive tournament exit predictor with speculative updates, branch/call target buffers, branch type and return predictors,
RT	Four 32-register banks, one each for 4 SMT threads, inter-block register dependence check logic.
IT	16 KB bank, 64-byte lines, one cycle hit latency,
DT	2-way, one-cycle 8 KB L1 cache bank, with 64-byte cache lines, cache-line interleaving among 4 DTs, one 256-entry LSQ, one-entry coalescing write buffer, 16-entry, fully-associative data TLB, MSHRs supporting up to 16 requests to up to four cache lines, memory-side, 1024-entry, single-bit dependence predictor to predict the dependences between program stores and loads.
ET	64-entry reservation station holding decoded instructions a five-stage, single-issue execution pipeline, one integer unit and one FP unit, single cycle basic integer operations, 3-cycle, pipelined integer multiply, 24-cycle, non-pipelined integer divide, 4-cycle FP operations, no support for FP divide and sqrt
Processor	30 tiles: 1 GT, 4 RTs, 5 ITs, 4 DTs, 16 ETs 16-wide issue, 1024-entry window

Table 4.2: Composition of the processor tiles.

the dissertation explores mechanisms for improving hard and soft error reliability by taking advantage of these synergies.

4.3.1 Fault Avoidance

The primary concern of this dissertation is with architectural techniques for improving reliability, and since fault avoidance is typically implemented at the device or circuit level it is not our primary focus. However, we do believe that the principles behind distributed architectures provide some important advantages for fault avoidance.

Borkar et al. propose a design shift from maximizing transistor density to a regular design fabric optimized for balanced design as a key enabler for gigascale integration [6]. A **modular design** approach that exploits concurrency reduces the emphasis on clock frequency and thus is less susceptible to extrinsic and intrinsic failures that lead to parametric or delay faults [6]. It can also aid in distributing the power consumption more evenly across the chip area, reducing the severity of hotspots that accelerate these failures [31]. **On-chip networks** constrain the wiring layout and naturally achieve a more regular interconnect fabric, analogous to metal patterning and design rule constraints that can be applied at the layout level [6]. This can significantly reduce the wiring congestion in the design and the susceptibility to design rule induced systematic defects. Static scheduling of block instructions can optimize for load-balancing across the ETs and the network links as an additional constraint, to reduce the possibility of highly skewed thermal and power densities.

It is more difficult to enumerate the advantages of distributed architectures for soft error avoidance, since techniques such as SOI and radiation hardening are typically applied at the device or circuit level [9, 20]. Overall, since distributed architectures reduce the emphasis on clock frequency, they allow for more slack in the hardware logic paths that can be exploited to implement circuit techniques such as appropriate latch selection or selective node engineering to reduce soft error susceptibility at low overhead [11, 88].

4.3.2 Fault Tolerance

Chapter 1 referred to the four necessary and sufficient principles of fault tolerance proposed by White et al. [14]: a) redundancy, b) fault detection and annunciation, c) fault isolation, and d) on-line repair. This section discusses how the features of the TRIPS architecture help in achieving these four principles. For each of the three distributed architecture principles, we first discuss the synergy with hard error reliability, and subsequently discuss soft error reliability. We presently limit the discussion to improving soft error reliability through redundant execution. Further, we envision redundant execution of blocks in the TRIPS architecture, analogous to instructions in conventional architectures [91].

Modular Design: The tile-based modular design provides abundant redundancy in the architecture that can be exploited for defect tolerance. For instance, since there are sixteen ETs, the architecture may easily tolerate a defective ET without a significant drop in the execution resources. This is possible because the tile-based modular design effectively reduces the size of the *field replaceable unit* (FRU) thus enabling fine-grained reconfiguration [18]. Further, a tile also provides a natural granularity for fault detection, isolation, and repair using advanced BIST techniques [24, 116].

The key advantage that tile-based design provides for soft error reliability, is that it enables temporal and spatial redundancy to handle both soft errors and intrinsic failures which may appear transient in nature. Further, since all the queues in the TRIPS architecture are statically block-partitioned (as described in Section 4.2), redundant execution of TRIPS blocks also provides spatial redundancy within the queues.

Cooperative Hardware/Software Techniques: We make the key observation that the TRIPS block-atomic execution model can provide redundancy that can be exploited for defect tolerance. Redundancy in a TRIPS block exists whenever it is under-utilized and contains fewer than 128 instructions. While a block that contains 128 instructions requires the full capacity of the distributed instruction window and cannot tolerate even a single

defective ET, a block that is under-full grants the scheduler the capability to successfully accommodate the instructions on a subset of the ETs that are functional. The block-atomic execution model provides an ideal granularity for the scheduler to perform this fault-aware instruction scheduling. This software-assisted, fine-grained fault reconfiguration, is similar in concept to logical partitioning that enables coarse-grained dynamic CPU sparing in IBM pSeries systems [28]. Chapter 5 explores this technique in greater detail.

The redundancy available in under-utilized TRIPS blocks can also be used by the compiler to insert redundant instructions for soft error detection. Further, the compiler can enforce spatial redundancy, in addition to temporal redundancy, by appropriately scheduling the redundant instructions on spatially redundant ETs. Since TRIPS blocks provide a natural granularity for fault detection, they also enable a different redundant execution model that is independent of block utilization, where each block as a whole is redundantly executed. This block-granular redundant execution model also allows selective redundant execution, in which the hardware or software identifies a subset of the blocks for redundant execution to trade fault coverage for improved performance overhead [117–119]. Chapter 7 explores the block-granular redundant execution model, and selective redundant execution in greater detail.

On-chip Networks: Modular design using routers allows the architecture to isolate a defect to a particular router in a tile. On-chip networks also provide the potential for spatial redundancy in the paths between a given source and destination tile. This is possible because each tile along the path contains a dedicated router that can independently decide the next hop for a packet, unlike a global wire that travels directly from the source to the destination. While the degree of flexibility in the paths depends on the routing function, in general, it allows for fully-adaptive routing algorithms that are capable of routing packets around defective tiles [120, 121].

The OPN in the TRIPS architecture currently implements y-x dimension-order routing that specifies exactly one route for a given source and destination tile pair. Hence, it

does not provide the adaptivity that the hardware can use to exploit the inherent spatial redundancy in the paths for defect tolerance. However, since the distributed execution resources are exposed to the TRIPS compiler, this inherent redundancy can be exploited at the software level by intelligent fault-aware instruction scheduling that can avoid the defective tiles and communication paths. Chapter 5 explores compiler-assisted fault reconfiguration in greater detail, and also discusses the implications of augmenting the OPN to be fully-adaptive.

The key advantage that on-chip networks provide for soft error reliability, is that it provides temporal and spatial redundancy along the communication paths. While temporal redundancy is naturally achieved during execution, the TRIPS compiler can enforce spatial redundancy by scheduling the instructions of the redundant block to use spatially redundant paths that still follow y-x dimension-order routing. Of course, as mentioned above spatial redundancy can also be achieved by enhancing the routing function of the OPN to provide this adaptivity.

4.3.3 Fault Evasion

As described in Chapter 1, fault evasion techniques observe the behavior of the system for periods of high vulnerability to hard or soft errors, and trigger preventive reconfiguration to reduce the probability of occurrence of an error. Dedicated hardware or software can be added to the system to perform this introspection, and the symptoms monitored can either be direct indicators of an error that has already occurred, or indirect indicators that signal periods of high vulnerability to errors.

For hard error evasion, the advantages that the TRIPS architecture provides are exactly the same as those for fault tolerance; the only difference is that fault reconfiguration is performed preventively in this case. While IBM pSeries servers already employ fault evasive techniques by monitoring intrinsic failure rates [28], a tile-based modular design and use of on-chip interconnection networks enables more fine-grained reconfiguration and

adaptivity. The symptoms monitored may include existing failure rates, and hot spots due to high tile and link utilization, and congestion which can be used as a prognosis of future failure locations.

Prior research has shown that processor utilization is a good indicator of vulnerability to soft errors [122], and Mukherjee et al. refined this analysis and proposed the *architectural vulnerability factor* (AVF) as a measure of a structure's vulnerability to soft errors [123]. To the first order, the amount of time for which unprotected state is resident in a structure or its *occupancy* determines the likelihood of a soft error.

Modular Design: A tile-based design provides an opportunity to detect periods of spatial non-uniformity in soft error vulnerability across different tiles [124], and selectively trigger reliable execution mode only for the vulnerable tiles. A fault evasive technique can either reduce the electrical vulnerability of the tile by changing the operating conditions such as voltage and clock frequency, or use architectural load-balancing and throttling techniques to spread the vulnerability more uniformly across the tiles.

Cooperative Hardware/Software Techniques: The compiler or the hardware can either statically or dynamically monitor the factors that affect soft error vulnerability including but not limited to occupancy, and processor utilization, and use it to dynamically transition to a less vulnerable execution mode as described above. While performing this analysis at a full program granularity hides the time-varying behavior, performing it at a per-instruction granularity provides information that can be difficult to exploit in a timely fashion limiting the potential benefits. In Chapters 6 and 7, we show that the TRIPS block-atomic execution model provides an ideal granularity for performing such introspective analysis, because it can operate at the block boundary allowing enough time for the information to be effectively used during the block execution lifetime.

On-chip Networks: The TRIPS on-chip networks act as an enabler for the fault evasion techniques just described. As explained in Sections 4.2.3 and 4.2.5, block fetch, register read, execute, and commit are performed using multiple independent on-chip networks (GDN, OPN, GCN, GSN). We make the key observation that the occupancy of block state, and hence its vulnerability, can be regulated by controlling the relative timing of the operations on these different networks. For instance, implementing a just-in-time block fetch protocol on the GDN to trigger instruction fetch only upon receiving a block register input on the OPN, can potentially reduce the occupancy of block instructions in the ETs by fetching them just when necessary. Chapters 6 and 7 explore this observation in greater detail.

4.3.4 Performance-Reliability Tradeoff

For hard errors, the primary advantage of a distributed architecture is the opportunity for graceful degradation provided by the abundant inherent redundancy. Of course, detailed analysis is necessary for both hard and soft errors to remove single points of failure from the design that can render the full chip defective [26]. The TRIPS architecture provides many opportunities to reduce the performance overhead of soft error detection using redundant execution of TRIPS blocks.

Modular Design: The block-partitioned modular design of the processor queues that service register reads, loads, and stores can be naturally extended to pass values speculatively from the primary execution of a block to its redundant execution. These values are then verified by the block's redundant execution before they are committed to architectural state. Decoupling operand delivery from verification overlaps the execution latencies of the primary and redundant block and potentially improves performance. Further, these queues also implicitly serve as synchronizing structures that control the relative slack between the primary and redundant execution streams.

Cooperative Hardware/Software Techniques: The block-atomic execution model provides several opportunities for reducing the performance overhead. While checking the output of every instruction detects faults in execution very early, it requires high bandwidth for output comparison [125]. Since only the block outputs have to be compared, the block-atomic execution model provides a favorable compromise between fault detection latency, and the bandwidth and storage requirements for output comparison. Prior research demonstrated that the block-atomic execution model reduces register file updates by 30-90% by using direct communication of temporary values between producing and consuming instructions within a block [50]. The redundant execution model described in detail in Chapter 7 allows multiple primary and redundant blocks to execute concurrently for higher performance. The model also exploits static knowledge of instruction placement to implement fine-grained, and light-weight communication of data between the primary and redundant blocks to further reduce performance overhead.

On-chip Networks: On-chip networks are the enabling technology for implementing fine-grained communication between the primary and the redundant blocks, that can reduce the performance overhead of redundant execution. The TRIPS architecture has several different on-chip networks for implementing different processor functions, and can be reused to communicate program instructions, commands, and data directly from the primary to the redundant block.

4.4 Summary

Table 4.3 summarizes the features of the TRIPS architecture that help in improving hard error reliability. As described in Section 4.3.3, the advantages that the TRIPS architecture provides for hard error evasion are exactly the same as those for hard error tolerance, so Table 4.3 only shows the synergies for hard error tolerance. Table 4.4 summarizes the features of the TRIPS architecture that help in improving soft error reliability. As described

in Section 4.3.4, the features of the TRIPS architecture also provides opportunities to reduce the performance overhead.

In this chapter, we qualitatively described in detail the synergy between the principles of distributed architectures and the approaches for improving reliability. Though we focus primarily on the TRIPS architecture, we recognize that these synergies apply to distributed architectures in general since they are built on the same core principles. Section 4.1.4 described how these principles are translated to specific mechanisms for three different distributed architecture design styles. For instance, these synergies would also apply to a CMP that uses modular design at the intra- and inter-processor granularity, on-chip networks in the memory hierarchy, and software threads for extracting concurrency. Prior research has explored hard and soft error reliability mechanisms exploiting these synergies on a CMP architecture [19, 126–128]. In the remaining chapters, we quantitatively evaluate specific mechanisms in the TRIPS architecture for hard and soft error reliability based on a subset of these synergies.

Synergy between TRIPS Features and Hard Error Tolerance		
Principles of Fault tolerance	**Feature of TRIPS architecture**	**Technique for improving reliability**
Redundancy	Tile-based modular design	Avoid defective tile
	On-chip networks	Avoid defective path, router
	Block-atomic execution model	Blocks with fewer than 128 instructions have redundancy
Fault detection	Tile-based modular design On-chip networks	Modular BIST engines at each tile and router
Fault isolation	Tile-based modular design On-chip networks	Reduced size of Field Replaceable Unit (FRU)
On-line repair	On-chip networks	Adaptive routing algorithms Routing tables for reconfiguration
	Static instruction placement Block-atomic execution model	Fault-aware instruction scheduling Ideal granularity for rescheduling

Table 4.3: Features of the TRIPS architecture that aid in implementing the principles of fault tolerance for hard error reliability

Synergy between TRIPS Features and Soft Error Tolerance		
Principles of Fault tolerance	**Feature of TRIPS architecture**	**Technique for improving reliability**
Redundancy	Tile-based design	Spatial, temporal redundancy, spatial redundancy in queues
	On-chip networks	Spatial redundancy on communication paths
	Block-atomic execution model	Insert redundant instructions in the block, if there are empty slots
	Static instruction placement	Compiler can enforce spatial redundancy
Fault detection and isolation	Block-atomic execution model	Natural granularity for fault detection and isolation at block boundary. Redundant execution of each block
On-line repair	Block-atomic execution model	Ideal granularity for checkpointing for fault recovery
Synergy between TRIPS Features and Soft Error Evasion		
Fault Evasion	**Feature of TRIPS architecture**	**Technique for improving reliability**
	Tile-based design	Exploit spatial non-uniformity in vulnerability by electrical or microarchitectural techniques Eg: non-uniform DVFS
	On-chip networks	Control GDN, OPN, GCN independently to reduce occupancy of block state and thus reduce block vulnerability
	Block-atomic execution model	Ideal granularity for introspective fault-evasive techniques to measure & reduce vulnerability

Table 4.4: Features of the TRIPS architecture that aid in implementing soft error reliability

Chapter 5

Techniques for Improving Hard Error Reliability

In this chapter, we explore techniques based on features of distributed architectures that can be used to improve chip yield and enhance the graceful degradation of fail-in-place systems. We observe that many structures even in current day chip designs, beyond just DRAM and SRAM, contain a substantial degree of regularity. For example, modern superscalar processors contain regular structures for managing register renaming, and for queuing instructions and other pending operations. Many processors contain parallel or replicated structures, such as execution units and set associative caches. We begin by showing that exploiting microarchitectural redundancy is a powerful technique, and can be effectively applied towards the current mainstream design methodology that uses chip multiprocessors composed of multiple dynamic superscalar processors, to achieve significant improvements in yield at future technologies.

Chapter 4 argued that future architectures will use greater software cooperation to exploit concurrency in a technology scalable manner, which naturally also suggests using software assistance for performing fault reconfiguration. In the second half of this chapter, we propose an innovative compiler-assisted fault reconfiguration technique for static

architectures like the TRIPS architecture, that exploits many of the synergies described in Table 4.3. The technique takes into consideration both the opportunities and the extra demands placed by important new features including on-chip interconnection networks, and the greater reliance on software.

Some of the redundancy, such as the replicated ETs in the TRIPS architecture, and extra entries in bookkeeping structures such as the instruction window, register renaming queues, and reorder buffer, can be removed with moderate performance penalty. Processors also contain non-essential structures, such as branch prediction tables, that enable higher performance but are not required for correct execution of programs. While modern chips are typically declared *functional* only if *all* of the components are fully functional, we propose that chips with some defective components are still useful and can improve overall yield or achieve higher availability in a fail-in-place system. Based on this idea, we extend *performance binning* to include a new method of differentiating chips that come off the fabrication line based on their relative performance, and propose a new yield metric called *performance-averaged yield* (Y_{PAV}) that is a function of the performance range of each bin and the number of chips in the bins.

We analyze these degraded chips in the context of improving yield for current and future technologies. We further explore different granularities of degraded components, from processor sub-components to whole processors, for uniprocessor chips and future architectures based on chip multiprocessors [101]. We explore dynamic mechanisms for fault reconfiguration in dynamic superscalar architectures, and fault-aware instruction scheduling heuristics in the statically scheduled TRIPS architecture. Our results indicate that the Y_{PAV} for a dynamic superscalar processor can be improved from 85% to 98% with certain assumptions about defect density and defect size. For chip multiprocessor architectures at future technologies, we show that microarchitectural redundancy provides substantial benefits achieving Y_{PAV} of up to 99.6%. We argue that compiler-assisted fault reconfiguration is more efficient than pure hardware based mechanisms for static architectures such

as TRIPS, and propose fault-aware scheduling heuristics which, for a set of EEMBC and SPEC benchmarks successfully reschedule the program to avoid defective resources with less than 4% loss in performance. We also discuss hybrid hardware-software techniques for fault reconfiguration as an interesting avenue that is worth investigating in the future.

These results show that many random manufacturing defects can be tolerated by exploiting microarchitectural redundancy. These results also suggest that augmenting fail-in-place systems with chip-level diagnostics and reconfiguration could provide more graceful degradation when components become faulty. Section 5.1 introduces the notion of performance-averaged yield and describes how it can be calculated. Section 5.2 applies the performance-averaged yield to dynamic superscalar architectures in the context of both a uniprocessor and a multiprocessor. Section 5.3 extends fault reconfiguration to the compiler and evaluates fault aware instruction scheduling heuristics. Section 5.4 summarizes the related work, and finally Section 5.5 presents our conclusions.

5.1 Performance-Averaged Yield

Today, it has become common for manufacturers to separate chips that are for sale into speed bins based on their operating frequency and to offer them for sale at different prices. More recently, some have made use of a more general *performance binning* strategy that separates parts into bins of guaranteed performance levels rather than bins based solely on operating frequency [129]. We propose that designs that include replicated or non-essential functions in support of increased performance be enhanced with the capability to disable some of these structures in face of defects detected within the circuitry. Chips of different end-performance, corresponding to different degraded configurations, can be offered at different prices, extending the current manufacturers use of speed binning. We formalize this notion of performance binning, and propose a new yield metric called *performance-averaged yield* (Y_{PAV}) in which the total yield is a function of the performance range of each bin and the number of chips in the bins.

Yield conventionally includes only chips that are fully functional, taking into account redundant rows to increase yield in caches. We propose to augment chips with specific fault reconfiguration capabilities, so that non-functioning components that fall within the domain of these capabilities can be successfully disabled to produce a functional chip, albeit with some performance degradation. While a fully functional processor with all of its components having their full capacity achieves peak performance, the overall yield ($Y_{OVERALL}$) that includes both fully functional processors and processors with some components disabled, exhibits a range of different system end-performance values. However, a fair methodology for calculating yield must now account for the system end-performance while combining the yield of fully functional processors and processors with performance degradation.

While $Y_{OVERALL}$ treats all the processor configurations equally regardless of their degraded state, *performance-averaged* yield (Y_{PAV}) specifically aims to differentiate between fully functional chips and chips with degraded components. Our design of the Y_{PAV} metric achieves this by using the performance of the resulting chip configuration as the discriminating measure. Using this formulation captures both the effects of redundancy—improvements in yield and reductions from peak performance. Processor companies typically use a carefully selected suite of applications, that are representative of the target market segment, to calculate processor performance, so that it can be easily compared with other designs. We propose to measure system performance in instructions-per-cycle (IPC), which is a standard metric used for measuring processor performance. Adding three steps to the algorithm for computing $Y_{OVERALL}$ gives us Y_{PAV}. First, the $MAXIPC$ corresponding to the base configuration (which is the fully functional processor configuration) is calculated. Each degraded configuration is then associated with a *relative IPC*, which is the ratio of its IPC to the $MAXIPC$. Finally the yield of each degraded configuration is scaled by its *relative IPC* and accumulated to give Y_{PAV}. This is described by the equation:

$$Y_{PAV} = \sum_{i \in \ all \ configurations} Y_i \times \frac{IPC_i}{MAXIPC} \tag{5.1}$$

5.2 Exploiting Microarchitectural Redundancy for Defect Tolerance in Dynamic Architectures

In this section, we propose to extend the *design for yield* techniques in the microarchitectural level by specifically identifying the redundancy in different on-chip components and the mechanisms by which it can be exploited. We suggest that by allowing some of these redundant structures to be disabled or degraded in the face of internal defects, the chip can offer improved defect tolerance.

We present results for yield enhancement at future technologies and chip microarchitectures as a function of the defect characteristics and the redundancy models. We examine the yield trends for a uniprocessor with and without redundancy, analyze the benefits of microarchitectural redundancy for chip multiprocessor architectures, and finally end with a comparison of the yield improvements attained using different redundancy models. We consider the two main chip topologies proposed in Chapter 2; a uniprocessor with a constant architecture, whose area decreases rapidly with decreasing feature size because the microarchitecture is kept constant, and a chip multiprocessor architecture which includes increasing number of processors per chip with decreasing feature sizes, and hence maintains constant chip area. Chapter 2 described the modeling infrastructure that integrates a basic yield model and a microprocessor area model with the redundancy model of the chip components, in detail. This overall chip yield model is extended to accommodate newer on-chip redundancy models proposed to achieve higher defect tolerance. Since there can be a performance penalty depending on the degraded configuration, we use a microprocessor simulator to measure the chip end-performance across the range of different configurations.

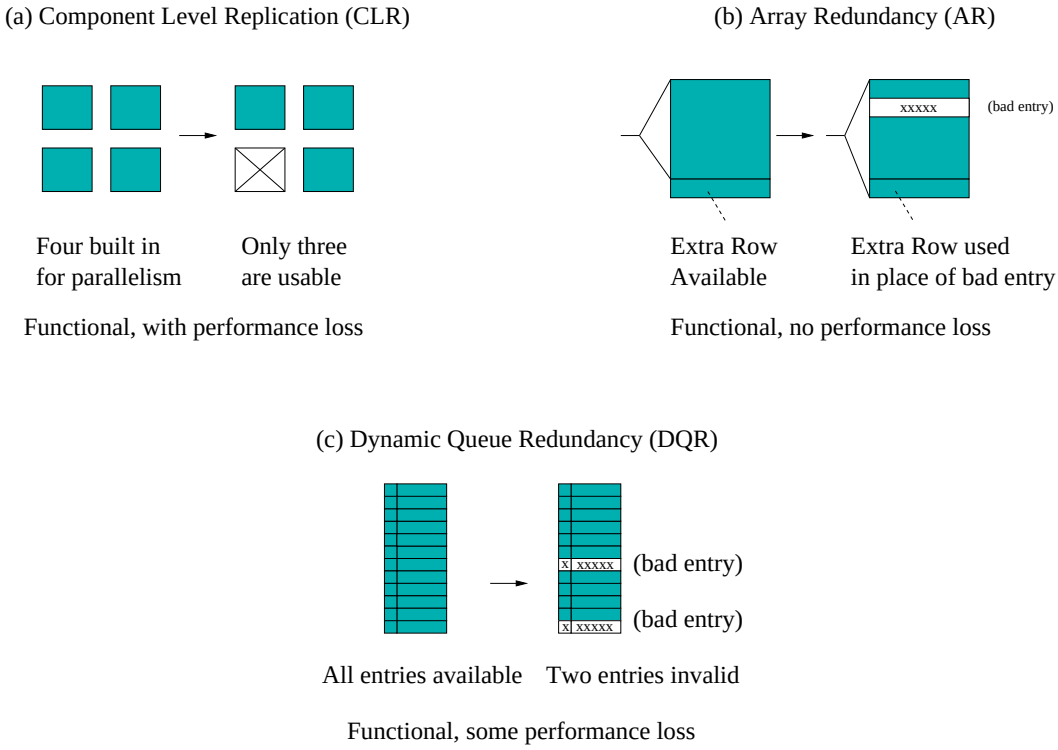

Figure 5.1: Basic redundancy models

5.2.1 On-chip Redundancy Model

This section describes each redundancy model and our implementation of the redundancy models in the different processor components. In the future, chips will likely contain multiple processors, when we can imagine a set of *intra-processor* redundancies as well as *inter-processor* redundancy at the next level of hierarchy. As designs strive for more outstanding operations in flight, it is likely that this potential for redundancy will increase over time. As a basis for analyzing the effects of the different redundancy types, we use the same processor model shown in Figure 2.3 of Chapter 2, that is similar to Alpha 21264 [43]. Both the integer and floating point clusters are symmetric and each have 2 functional units within them. The processor also has an on-chip L2 cache of 1MB. We identify three primary types

of redundancy as a basis for our redundancy model. In the next few paragraphs we explain each type of redundancy shown in Figure 5.1.

Component Level Redundancy (CLR): In CLR, the component is typically replicated to provide additional performance through parallelism, but only a subset is actually required for correct functionality. Each component that has CLR has a *resource* line associated with it, and the component's BIST module sets the *resource* line to be permanently BUSY in the event the component is disabled due to internal faults. The parent control logic already contains mechanisms that restricts its use each cycle to only those components whose resource lines are FREE. For instance, the instruction scheduling logic is implemented using wakeup arrays that contain *RESOURCE AVAILABLE* lines indicating which resources are FREE in the given cycle [130]. The execution clusters and the internal ALUs of the processor are covered by the CLR model. The hierarchical nature of the CLR for the clusters and ALUs provides coverage over the control logic of the individual clusters.

The Alpha 21264 [43], for example, has two integer clusters each with two integer arithmetic units. The CLR concept can be applied within a cluster since it can operate correctly with only one functional unit. The clusters are not exact copies of each other in the Alpha 21264, and the instruction dispatch logic is statically biased to distribute instructions between them. But we use a modified Alpha 21264 design where they are symmetric, and hence the CLR model can be applied here as well. In the future, as designers continue to struggle with managing the complexity of large designs, we expect this sort of *design for reuse* to become more common. Further, if CLR can be proven to have a positive effect on defect tolerance and yield, it is reasonable to expect that the amount of CLR is likely to increase over time.

Array Redundancy (AR): Array redundancy was already described in Chapter 2, but is included here for completeness. When defects are detected in rows or columns of bit cells in the main body of the array, the AR mechanisms can be configured to effectively

steer the decode towards the redundant entry rather than towards the bad row or column. This technique is already commonly used in many types of RAM chips as well as in the embedded RAM structures found in more general purpose chips such as microprocessors. From a yield perspective, AR is attractive because a relatively small investment in area can offer excellent defect tolerance for the entire structure. In many cases, this can drive the yield loss due to these structures to very low levels with no loss in performance. The set associative L1 and L2 caches are way-interleaved allowing both CLR (disabling one set) and AR (redundant row steering within one of the operational sets). The CLR for the caches provides coverage over the peripheral logic of the individual cache banks also. Consistent with the accepted design practice [48], the redundant rows and columns are about 2.5% of the base cache capacity. We also allow the configuration with no on-chip L2 cache, *ie.*, with both L2 cache banks defective. The TLBs are also covered by the AR model.

Dynamic Queue Redundancy (DQR): DQR is the third type of redundancy that we explore. A *valid bit* is added to each queue entry that has DQR. If a particular queue entry has defects, it can be permanently disabled by clearing the *valid bit*, thus decreasing the number of *available* entries. The existing protocols that add queue entries are modified to stall the machine when the *available* queue entries are full. Downstream queue access logic is also augmented so that the queue entries marked *invalid* are never processed. In highly pipelined designs, as well as designs that support dynamic reordering of operations, many structures such as the reorder buffer, the issue window, the register remappers, the load and store buffers are implemented as queues. For example, in the Alpha 21264, the reorder buffers, the issue window, the register remappers, the load buffers, and the store buffers are all implemented as queues. There is a minimum size to these queues to support basic functionality, and the larger sizes are intentionally used to gain performance. In our implementation of DQR, we include spare queue entries to provide some defect tolerance without losing any performance, similar to AR. Nevertheless, our experiments show that

Processor Redundancy Configuration			
Microarchitecture Resource	Base capacity / Spare entries	Redundancy Model	Minimum operational size
INT, FP Instruction Window	20 / 1	DQR	20
INT, FP Register File	80 / 2	DQR	80
INT, FP Map Table	32 / 1	DQR	32
Execution units per cluster (INT Alu, FP Alu, INT Mult, FP Mult)	2 / 0	CLR	1
INT, FP Clusters	2 / 0	CLR	1
Reorder Buffer	80 / 2	DQR	80
Load / Store queue	32 / 1	DQR	32
TLBs (Fully associative)	128 / 2	AR	128
L1 I, D cache (2-way associative)	64KB / 1.5KB	AR, CLR	32KB
L2 cache on-chip	1MB / 24KB	AR, CLR	0MB

Table 5.1: Processor redundancy configuration

disabling one or two entries in most of these queues results in at most 1% loss in performance. All of the instruction window queues, register files, map tables, reorder buffers, and storage queues are associated with the DQR model.

Table 5.1 summarizes the redundancy model in the different processor components. The *spare entries* provided in the components are used only in the face of defects and do not contribute to additional performance. Elements of the processor not listed in the table, like random control logic, and logic that is used to implement the redundancy model itself, have no redundancy coverage in our example design. Nevertheless, approximately 85% of the total area of the processor has coverage through redundancy, as compared to 50% with AR alone in the L1 and L2 caches. This configuration and aggregate model is used for the uniprocessor and multiprocessor yield analysis described throughout this study. As described in Section 2.3.2, baseline yield (Y_{BASE}) corresponds to a processor with AR in the L1 and L2 caches.

5.2.2 Extensions to the Overall Chip Yield Model

Section 5.2.1 describes how a single processor component may have more than one form of redundancy. If the multiple redundancies are non-hierarchical in nature, the overall component yield is simply equal to the product of the yield of the individual regions with the different redundancy models. The individual yields can be composed using a simple product because the Poisson model treats defects as completely independent. On the other hand if the component has redundancies that compose hierarchically, we begin by applying the method at the lowest level at which the redundancies of the regions are non-hierarchical, and then reapply the method recursively at each higher level of hierarchy. The chip area model is used to estimate the area of the different component regions. The redundancy model of a region is one of CLR, AR, DQR or *no redundancy*. The Poisson Yield model is used directly to calculate the statistical yield of a region with no redundancy. The method to calculate the yield of a region with one of the three primary redundancy schemes is described below.

A redundancy model specifies the minimum number of working entries the component must possess to ensure correct overall functionality. The component yield is then simply the probability that it has at least its minimum subset of entries working out of the total number of entries including the spares. The overall component yield is therefore the sum of the probabilities associated with all the configurations in which the component has at least the minimum number of working entries, out of the total number of entries including spares. The Y_R from this calculation is summarized using the well known binomial expansion:

$$Y_R = \sum_{i\,=\,mins}^{(bs+se)} C_{mins}^{(bs+se)} \times \mathrm{Pr(Good)}^{i} \times \mathrm{Pr(Bad)}^{(bs+se-i)} \tag{5.2}$$

where C_r^n is the *combinations* operator, $mins$ is the subset of entries required for correct functionality, bs represents the base number of entries in the component, and se

is the number of spare entries. The probability of a entry being functional or invalid is computed using the Poisson Yield model. For example, caches that have AR are provided with enough redundant rows and columns to greatly improve yield and at the same time show no reduction from peak performance. Hence in this case $mins$ becomes equal to bs, and the value of se is dependent on the cache capacity. Components with DQR also have very similar specifications. With CLR in the clusters, the processor could potentially have a configuration with only one functional cluster, in which case se is equal to zero and $mins$ is equal to one. Hence $Y_{OVERALL}$, not only includes the traditionally accounted fully functional chips but also includes chips with degraded components. The minimum subset of entries for each on-chip component determined by the specific redundancy model is given in Table 5.1.

5.2.3 Results

To evaluate the performance of the various degraded configurations we used the sim-alpha simulator which models the Alpha 21264 core in detail [131]. First, we configured sim-alpha to resemble our processor model. We further made modifications that enable us to simulate the different degraded configurations by selectively disabling on-chip components. Table 5.2 shows the seven benchmarks we chose from the SPEC2000 benchmark suite and *sphinx*, a speech recognition benchmark, to provide a wide range of behavior in their usage of the memory system and the execution resources. The applications *mesa*, *equake*, *eon*, and *gzip* show relatively high IPC and are more sensitive to the available execution resources. The applications *sphinx*, *mcf*, *swim* and *art* are memory intensive in nature and show greater sensitivity to cache capacities. For each benchmark, we simulated the sequence of instructions which capture the representative phase of the program, determined by using SimPoint [132]. Table 5.2 also shows the number of instructions skipped to reach the start of the execution phase ($FFWD$), the number of instructions simulated (RUN), determined using SimPoint [132], and the maximum IPC for each benchmark at the base

	Benchmark category	Benchmark name	FFWD (x100M)	RUN	MaxIPC
INT	Memory intensive	181.mcf	336.3	100M	0.13
		sphinx	60	200M	0.57
	Processor bound	164.gzip	332	100M	1.76
		252.eon	207.3	100M	1.29
FP	Memory intensive	171.swim	1196	100M	1.02
		179.art	66.3	100M	0.26
	Processor bound	183.equake	193.4	100M	1.11
		177.mesa	639.9	100M	1.34

Table 5.2: Benchmarks used for performance experiments

configuration.

Two important factors contribute to Y_{PAV} being nearly equal to $Y_{OVERALL}$. To describe the factors we plot the relative IPC distribution in Figure 5.2 for all the allowed processor configurations, not accounting for their actual yield or likelihood of occurrence. A subset of this data along with the degraded processor configuration is also shown in Table 5.3. As a review, the relative IPC is calculated as the ratio of the IPC of the degraded configuration to the peak IPC corresponding to the fully functional configuration.

The top portion of Table 5.3 shows the harmonic mean relative IPCs of a small subset of chip configurations having relative IPC greater than 0.8. In the bottom portion, the relative IPC drops below 0.8, with the last row corresponding to our most degraded configuration. First, the graph shows that 80% of the configurations have a relative IPC greater than 0.8. The remaining configurations having relative IPC around 0.55 correspond to the chips with a fully defective L2 cache. Second, there is enough redundancy in our processor model that most of the yield is also concentrated in configurations with high relative IPC, and highly degraded configurations such as in the bottom portion of the table never occur and hence provide no contribution to yield. Hence in all of the product terms contributing to Y_{PAV} (Equation 5.1) with non-zero yield (Y_i) the associated relative IPC is close to one.

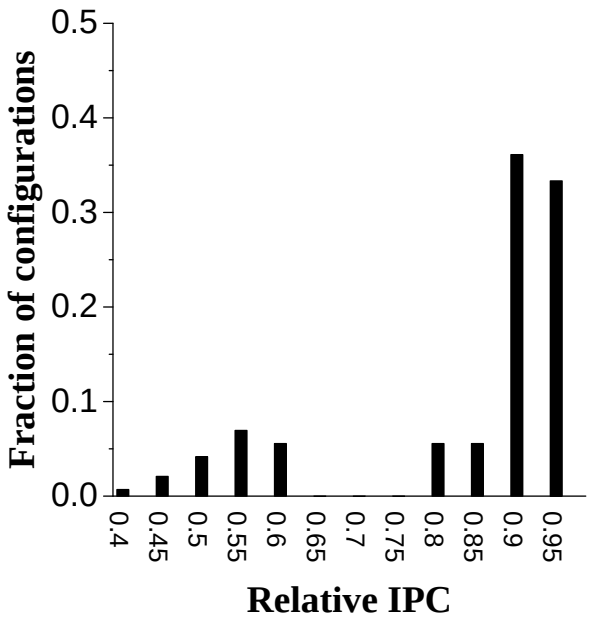

Figure 5.2: IPC distribution for the different configurations

Based on this analysis, we present our results for yield enhancement at future technologies and chip microarchitectures as a function of the defect characteristics and the redundancy model. We model the same constant-architecture uniprocessor and constant-area chip multiprocessor topologies proposed in Chapter 2. We first proceed to examine the yield trends for the uniprocessor with and without redundancy. We then analyze the benefits of microarchitectural redundancy for chip multiprocessor architectures, and finally end with a comparison of the yield improvements attained using different redundancy models.

Constant-architecture Uniprocessor Yield

Figure 5.3 shows Y_{PAV} obtained by incrementally adding different flavors of on-chip redundancy to the constant-architecture uniprocessor model. For instance, at 100nm the maximum contribution comes from L2 bank level redundancy, and CLR in the functional units dominates among all the other types of redundancy, which together increase Y_{PAV}

Degraded Processor Configuration					
Integer Functional Units	Floating Point Functional Units	I-Cache Capacity (KB)	D-Cache Capacity (KB)	L2 Cache Capacity (MB)	Relative IPC
4	4	64	64	1	1.0
3	4	64	64	1	0.97
2	4	64	64	1	0.93
4	3	64	64	1	0.99
4	2	64	64	1	0.98
4	4	32	64	1	0.97
4	4	64	32	1	0.97
4	4	64	64	0.5	0.94
2	2	64	64	1.0	0.93
2	2	32	32	0.5	0.85
1	1	64	64	1	0.73
1	1	32	32	0.5	0.69
4	4	64	64	0	0.65
2	2	32	32	0	0.5
1	1	32	32	0	0.44

Table 5.3: Relative IPCs for a few sample degraded configurations

to 98.8%. Since most of the configurations lie within 20% of maximum performance (see Figure 5.2) $Y_{OVERALL}$ (indicated by the dotted line), which includes both fully functional processors and processors with some components disabled, is 99.2%, which is only 0.4% greater than Y_{PAV}. Across technologies, Y_{BASE}, the yield with AR in the L1 and L2 caches, increases from 85.4% to a maximum of 93.7% because the gain from the rapidly decreasing chip area outweighs the increased susceptibility to yield loss due to the higher kill ratio. Second, the contribution of L2 bank level redundancy continues to be significant, and all the other types of redundancy give progressively diminishing returns. This is because the L1 and L2 caches occupy almost 70% of the chip area and the absolute area occupied by the remaining components becomes vanishingly small at smaller feature sizes. Finally, Y_{PAV} increases from 98% at 250nm to 99.2% at 50nm, and since most of the configurations lie within 20% of maximum performance (see Figure 5.2) $Y_{OVERALL}$, indicated by the dotted line, is at most 0.4% above Y_{PAV} across all technologies. The above result is

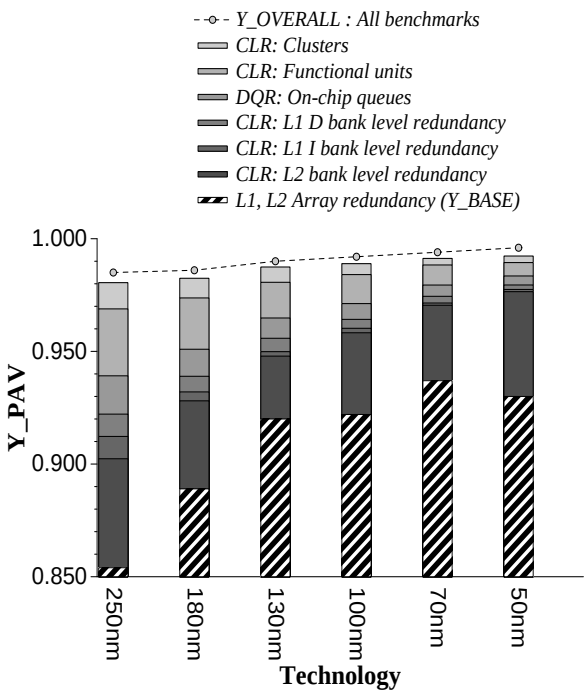

Figure 5.3: Yield for a constant-architecture uniprocessor model at normal defect size

significant because it shows that even though Y_{BASE} improves with technology, Y_{PAV} can be further improved by adding microarchitectural redundancy.

Constant-area Chip Multiprocessor Yield

In this study, we explore two types of multiprocessor redundancy. In intra-processor redundancy, a chip can have its processors in any of the allowed internally degraded states, but the entire chip is considered bad once the available redundancy is exhausted in even one of its processors. On the other hand a processor in a chip with only inter-processor redundancy becomes useless if any fault resides in it. However, if enough of the remaining processors are functional, the chip can still be operational. In this study, we consider the chip to be functional as long as there is at least one good processor per chip. However, due to the abundant redundancy available, in practice our models never produce chip configu-

rations with more than two bad processors per chip. Since we model the multiple threads to be independent, we calculate chip performance as the aggregate performance of all the cores on the chip. The algorithm for calculating Y_{PAV} from Section 5.1 can be naturally extended to a multiprocessor by modifying each step to account for the IPC of the entire multiprocessor (whether the configuration is fully functional or degraded). Extending the equation for Y_{PAV} from Section 5.1 to a multiprocessor:

$$Y_{PAV} = \sum_{j\ =\ all\ configurations} Y_j \times \frac{\sum_{i=1}^{N_c} IPC_{ij}}{(N_c \times MAXIPC_{core})} \tag{5.3}$$

where, N_c is the number of cores, IPC_{ij} is the performance of each core in that configuration, and $MAXIPC_{core}$ is the peak IPC of a fully functional processor. Since the atomicity of failure in the inter-processor redundancy model is a whole processor, a core has at most two states corresponding to $MAXIPC_{core}$ or zero IPC. The expression for Y_{PAV} can then be simplified to:

$$Y_{PAV} = \sum_{j\ =\ all\ configurations} Y_j \times \frac{N_{wj}}{N_c} \tag{5.4}$$

where, N_w is the number of fully functional processors in that degraded configuration.

Yield with Intra-processor Redundancy: Figure 5.4 plots Y_{PAV} across all technologies, obtained by incrementally adding redundancy to each processor in a multiprocessor chip with intra-processor redundancy. The x-axis shows the feature size and the number of processors per chip at each technology. At any given technology adding redundancy improves Y_{PAV} substantially. CLR in the functional units give maximum yield benefit, and the benefits from L2 bank level redundancy, DQR in the queues, and CLR in the clusters are comparable. For instance, at 70nm adding redundancy dramatically improves Y_{PAV}

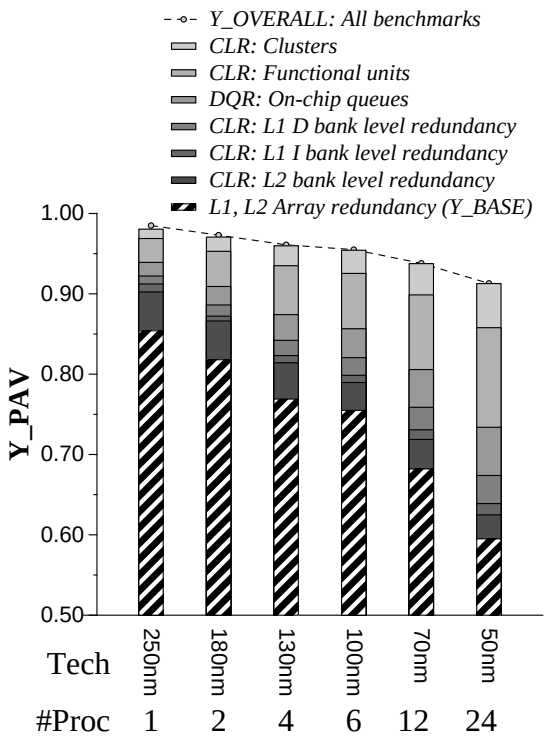

Figure 5.4: Yield with intra-processor redundancy at normal defect size

from 68.2% to 93.7%.

Three interesting features can be observed across technologies. First, the Y_{BASE} decreases substantially from 85.4% at 250nm to 59.5% at 50nm, because the kill ratio increases considerably at smaller feature sizes. Second, the instances of intra-processor redundancy on the chip increases linearly with the number of processors, and as a result the addition of redundancy leads to greater improvements in yield at smaller feature sizes. For instance, at 180nm Y_{PAV} increases by 4.4% with CLR in the functional units, whereas it increases by 12.4% at 50nm. Third, the higher yield benefits, depending on the redundancy model, imply that more chips are degraded at smaller technologies. But as the area occupied by a single processor decreases, its Y_{BASE} increases (see Figure 5.3), and hence the probability of it being defective decreases. Combined with the increasing number of processors

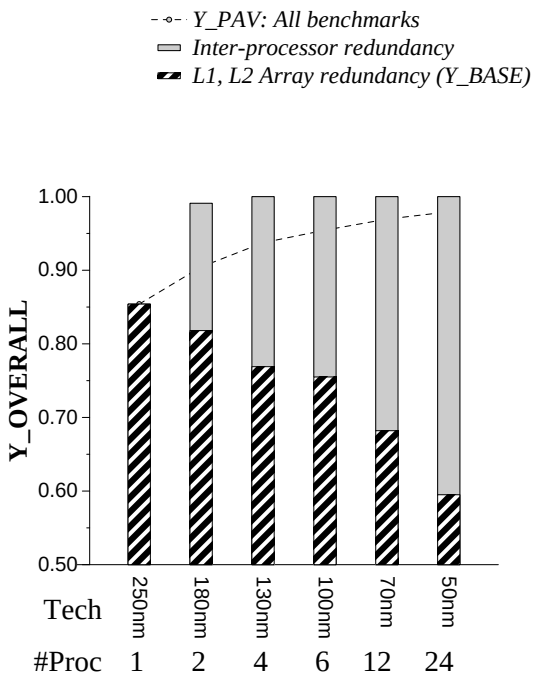

Figure 5.5: Yield with inter-processor redundancy at normal defect size

per chip, the fraction of degraded processors per chip decreases with technology. Hence, even though the number of degraded chips increases at smaller technologies, each resulting degraded chip configuration contains a majority of fully functional processor cores and very few degraded processors. As a result, Y_{PAV} continues to be within 0.2% of $Y_{OVERALL}$ at all technologies. Although there are significant benefits from adding redundancy, Y_{PAV} with all the types of redundancy drops from 98% at 250nm to 91.3% at 50nm due to higher kill ratio.

Yield with Inter-processor Redundancy: Figure 5.5 plots Y_{PAV} for a multiprocessor with inter-processor redundancy. Inter-processor redundancy gives coverage over the entire area of the chip and hence dramatically improves $Y_{OVERALL}$, approaching 100% at technologies beyond 180nm. The yield at 250nm alone is low because only one processor resides on the chip, which amounts to having no redundancy at all. Also recall that

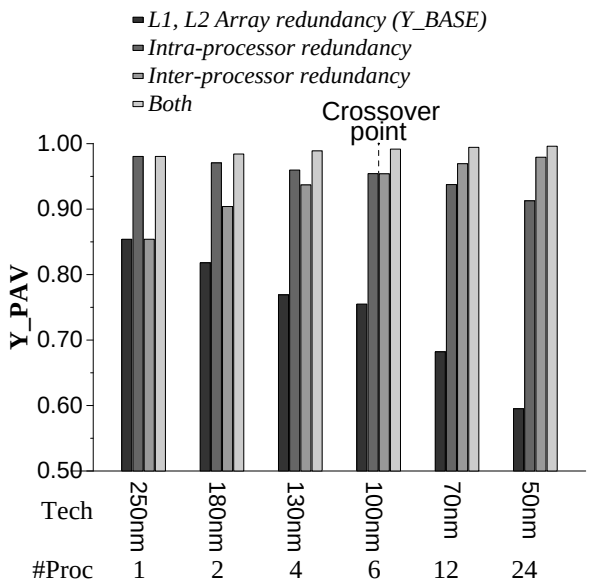

Figure 5.6: Comparison of Y-PAV for different redundancy models

the fraction of degraded processors per chip decreases with technology. As a result, Y_{PAV} increases uniformly from 85% at 250nm to 98% at 50nm.

Comparison of Redundancy Models: Figure 5.6 compares Y_{PAV} obtained using four different redundancy models. With only AR, Y_{PAV} decreases rapidly from 85.4% at 250nm to 59.5% at 50nm. Having intra-processor redundancy alone achieves high Y_{PAV}, which decreases slightly from 98% at 250nm to 91.3% at 50nm. Inter-processor redundancy gives coverage over the entire area of the chip and hence Y_{PAV} increases uniformly from 85.4% at 250nm to 98% at 50nm. The yield benefits offered by intra and inter-processor redundancy crossover at 100nm because of the opposite trends in their Y_{PAV} across technologies. A comparison of the improvements offered by intra and inter-processor redundancy models shows that, down to 100nm Y_{PAV} obtained using intra-processor redundancy is higher than from inter-processor redundancy after which we see greater benefits from the inter-processor redundancy model. The crossover point is dependent on the defect parameters

and the processor granularity at each technology. While this analysis assumes a constant defect density across technologies, larger defect densities will shift the crossover point to the right because the fault susceptibility per unit area of silicon increases, and hence fine grained redundancy becomes more appropriate. Also, while our CMP design is composed of a number of relatively small Alpha 21264-like cores, future CMP designs may take advantage of much larger uniprocessor cores to achieve technology scalable high performance over a wide range of applications [111]. Consequently, there will be fewer processors and significantly greater intra-processor redundancy than inter-processor redundancy per chip, which will again shift the crossover point to the right. Since intra and inter-processor redundancy offer different types of coverage, having both intra and inter-processor redundancy provides consistently high Y_{PAV} ranging from 98% at 250nm to 99.6% at 50nm, with a maximum improvement in Y_{PAV} of 3.75% over having only one of the types of redundancy.

These results show that exploiting microarchitectural redundancy is a powerful technique that can achieve significant improvements in the yield of dynamic superscalar CMP architectures at future technologies. However, as described in Chapter 4, future distributed architectures will use greater software assistance to achieve high performance. Architectures may adopt fully static or hybrid execution models that expose the distributed resources to the software for static allocation to achieve technology scalability. While error-free execution can be achieved in a dynamic superscalar processor by forcing the program to utilize only the functional processor resources, just disabling a statically allocated resource in a static architecture will lead to incorrect execution. Further, configuring a hardware component to be offline or online in a dynamic architecture, is relatively easy, and can be done using busy lines, free lists, or valid bits that mostly already exist, as described in Section 5.2.1. On the other hand, in a static architecture, either the application also has to be appropriately recompiled to account for the degraded configuration, or dedicated hardware resources need to be added to perform the necessary fault reconfiguration oblivious to the

software.

5.3 Fault-Aware Instruction Scheduling Heuristics for Static Architectures

In this Section, we propose techniques for compiler-assisted fault reconfiguration in static architectures, and identify potential advantages over purely hardware based mechanisms for defect tolerance. First, we propose that future static architectures push dynamic fault reconfiguration within the boundaries of a single processor to achieve greater yield and system availability. Second, we propose that the defective processor configuration be exposed to the compiler which can then perform efficient fault reconfiguration by intelligent instruction scheduling. We argue that fault-aware physical layout of the instructions can more effectively exploit the available spatial redundancy, and with fewer overheads than a purely hardware based approach to achieve both better performance and yield. We evaluate our mechanisms on the TRIPS architecture.

5.3.1 Design Space for Fault Reconfiguration in Static Architectures

The design space for fault reconfiguration in static architectures can be organized under three axes:

Granularity of reconfiguration: Fault reconfiguration within a chip can be coarse-grained and operate at a processor granularity, exploit fine-grained intra-processor redundancy, or both. Section 5.2.3 discussed the tradeoffs involved, and concluded that supporting intra-processor fault reconfiguration is beneficial either if the defect density is large, or if the cores that compose the chip are large.

Hardware or software based reconfiguration: Fault reconfiguration can be performed either using hardware techniques that are transparent to the software, or with software as-

sistance. In static architectures like TRIPS, in which the hardware configuration is already exposed to the instruction scheduler for spatial instruction placement (see Section 4.2), software-assisted fault reconfiguration is a natural proposition and provides interesting opportunities and challenges.

Economy of mechanisms: To be complexity-effective, fault reconfiguration must ideally reuse mechanisms that already exist, or propose new mechanisms that can be useful for achieving multiple goals. A lot of research has been done on hardware techniques for architectural adaptation of processor resources to achieve improved power and energy efficiency [105, 133]. Many of these core concepts and mechanisms can be efficiently adapted for fault reconfiguration.

Software based dynamic reconfiguration on logically partitioned IBM pSeries symmetric multiprocessor systems allows movement of hardware resources from one partition to another enabling autonomic system management to optimize performance, resource utilization, and reliability [28]. It provides the foundation for self-healing and diagnosing software for dynamic CPU sparing that allows systems to transparently replace a defective processor with a fully functional processor with no impact on the application. This technique uses virtualization to manage system reconfiguration at the processor granularity, and can be naturally applied to future system-on-a-chip and CMP designs [126].

The Transmeta Crusoe processor pushed this technique within the boundaries of a single processor, and virtualized an X86 CPU by implementing a code morphing software layer that dynamically translated instructions from the target X86 ISA to the VLIW host ISA [134]. Traditionally, VLIW processors expose the processor pipeline details to the compiler, requiring all existing binaries to be recompiled following any change in the pipeline microarchitecture. The Transmeta processor solved this problem by exposing the actual VLIW hardware configuration only to the code morphing software that does the dynamic translation. Thus, only the translation software requires modification whenever there is a change in the underlying hardware. In general, virtualization can be used to isolate

the details of the hardware from the software whenever the underlying hardware configuration is expected to change, thus enabling application portability to new environments and processors.

5.3.2 Fault Reconfiguration in the TRIPS Architecture

In the context of the TRIPS architecture, the instruction scheduler already examines the entire distributed execution substrate, and accounts for all the constraints that contribute to optimal block performance (see Section 4.2.2). The scheduler can therefore naturally act as the virtualization layer, and take advantage of the abundant redundancy within the TRIPS processor to expose only the functional resources to the scheduling algorithm. Table 4.3 in Section 4.4 summarized the salient features of the TRIPS architecture that can be exploited for hard error reliability. The next few paragraphs discuss how the features of each TRIPS tile influence the choices for fault reconfiguration available in each axes of the design space.

Register Tile (RT): Defects in entries of the architectural register file, and the read and write queues can be tolerated using DQR described in Section 5.2.1. Further, with modest extra reconfiguration logic, the hardware can use one of the other four register banks provisioned for use in multi-threaded mode, if a defect in the random logic portion renders an entire bank of the register file unusable. Of course, this would mean that the processor can support only up to three threads instead of four. Similarly, since the read and write queues are statically block-partitioned, adding simple microarchitectural support to limit the maximum number of speculative blocks below eight in single-threaded mode, and the maximum number of threads below four in multi-threaded mode, allows an entire block partition to be disabled for even higher defect tolerance. Since the RT area is dominated by storage, we believe that these techniques will provide very high defect tolerance. However, if a defect renders the entire RT unusable, a program must be able to confine its usage to the functional RTs. Since the architectural registers are statically interleaved across the four RTs in a round robin fashion, register allocation must be made fault-aware to use only the

architectural registers in the functional RTs. While this technique is feasible at static compilation time to improve chip yield, redoing global register allocation at run-time to improve lifetime reliability is an expensive and inconvenient operation. Further, if a defect occurs in the router (OPN), or other communication logic (GSN, GCN, GDN), redundant logic and routing paths must be provided to allow communication between the different RTs.

Data Tile (DT): The data caches, the storage structures in the LSQs, and the miss-handling logic can be protected effectively using a combination of AR and DQR described in Section 5.2.1. Further, if the structures are sub-banked they can also use CLR. Since the DT area is also dominated by storage, we believe that these techniques will provide very high defect tolerance. However, if a defect in unprotected random logic disables an entire DT then no data accesses can be allowed to that tile. Since addresses are statically interleaved at a cache line granularity across the four DTs in a round robin fashion, the interleaving policy must be made configurable to use only the functional DTs. The tradeoffs associated with this technique are discussed in [26]. Further, if a there is a defect in the OPN router that disables communication through the DT, the load and store instructions that generate the data tile accesses, and the target instructions that consume the data must only be scheduled in rows or columns such that the y-x dimension-order paths in both cases do not include the defective tile. Alternatively, redundant logic and routing paths for all the networks (OPN, GSN, GCN, DSN) must be provided to allow communication between the different DTs.

Instruction Tile (IT): The instruction cache arrays, and other queues can similarly be protected using AR, DQR, and CLR to achieve very high defect tolerance. A defect that disables an entire IT is arguably more difficult to manage, since TRIPS uses a fixed allocation policy wherein each IT is responsible for delivering instructions to the ETs in the same row. One option is to use compiler assistance to reduce the maximum allowed block capacity from 128 instructions, and scheduling the block on the ETs in the rows

corresponding to the functional ITs. Since the ETs corresponding to the defective IT now only contain NOP instructions, it can be ignored without affecting correctness, albeit with a corresponding performance overhead. However, redundant logic and routing paths have to be provided to allow communication between the different ITs.

Global Tile (GT): While the I-cache tag, and predictor arrays can be protected using AR, GT also contains a significant amount of state machines and control logic which must be protected by conventional redundancy techniques such as duplication. Since there is only one GT we believe that the overhead due to duplication will be reasonable.

Execution Tile (ET): The ET is unique in many respects. Sankaralingam et al. demonstrated that the ETs consume 28% of the overall chip area [108]. Thus, while the layout and routing density of logic is less than regular memory arrays, the large area occupied by the ETs make them susceptible to defects. Further, like the GT, ET area is dominated by control logic, both due to regular datapath structures in the functional units, and significant amount of random control logic in the decode, select, register read, execute, and writeback pipeline stages. The key difference however is that, unlike the GT, there are sixteen ETs providing a substantial amount of redundancy both for computation and communication.

Also, prior research has shown that the ALUs and the operand register file are the hottest structures on the chip making them most vulnerable to intrinsic failures [31, 135]. TRIPS uses a three-level register storage hierarchy composed of the architectural register file and the read queues in the RTs, and the operand buffers in the ETs, to optimize for capacity, latency, and bandwidth. Since the ET buffers, that form the lowest level of the hierarchy, are the most frequently accessed due to their low latency and high bandwidth, they may potentially become a thermal hotspot in the design.

These unique aspects of the ETs make them the most interesting structure to study. Further, since the area of the other tiles (except GT) are dominated by storage, AR, DQR, and CLR can be effectively applied to achieve substantial defect tolerance. Hence we focus

only on the ETs in the rest of this study. We explore opportunities for using the instruction scheduler as the virtualization layer to exploit the fine-grained redundancy available. However, the design of the fault-aware instruction scheduling algorithm depends on the fault model for the ETs.

5.3.3 Fault Model

Based on the above analysis, we make the simplifying assumption that the yield of all chip components except the ETs is perfect, to help us with isolating the yield behavior of the ETs. This assumption allows us to calculate chip yield using the basic yield model proposed in Chapter 2, along with an accurate area model for the ETs alone. The area estimated by the model is within 1% of the actual area computed by synthesis of the ET RTL, thus validating the model. We also ensured that the area of the major individual components, such as the instruction and operand buffers, and the functional units, from the area model and synthesis matched each other closely. The redundancy model we assume for the components within an ET includes DQR for the instruction and operand buffers, but does not include CLR in the functional units because each ET contains only one functional unit of each type.

Table 5.4 shows the projected yield distribution for five technology nodes 180nm, 130nm, 90nm, 65nm, and 45nm. The number of ETs per chip increases with decreasing feature size to occupy a constant area of the chip. Since we assume that all chip components except the ETs have perfect yield, defects can only disable the ETs. Note that, as expected, the random defect limited yield of chips with all ETs fully functional is quite high between 84.2%-96.4%. More importantly, notice that the distribution of defective chips falls quite rapidly, with chips with one defective ET uniformly being the dominant common case at all technology nodes. Further, this result is independent of the chip configuration determined by the number of processors per chip and the number of ETs per processor. Based on this result, we use single-defective-ET as the fault model in the rest of this study, and focus on fault-aware scheduling heuristics that can tolerate one defective ET. Further, we consider

Defective ET Yield Distribution					
Technology node (nm)	180	130	90	65	45
Number of execution tiles per chip	8	16	32	64	128
Area (mm2)	43	45	43	45	43
Total number of chips	1000	1000	1000	1000	1000
Number of fully functional chips	964	948	928	888	842
Number of chips with 1 defective ET	35	51	70	106	145
Number of chips with 2 defective ETs	1	1	2	6	12
Number of chips with 3 defective ETs	0	0	0	0	1

Table 5.4: Determining the fault model using the yield distribution for the ETs

two specific variations of this general fault model:

- Faulty Node: The defect can occur anywhere on the ET except on the logic required for communication. This excludes the OPN router, and logic included for the GDN and GCN. So instructions cannot be scheduled on this ET, but the ET can be used as an intermediate hop between a different source and destination tile.

- Faulty Link: A more restrictive fault model which also allows defects in the operand router that transmits the results of computation to dependent instructions. Therefore, in addition to not scheduling any instructions, this ET also cannot be on any path between two functional ETs.

Figure 5.7 illustrates the two fault models, and the corresponding fault-aware in-

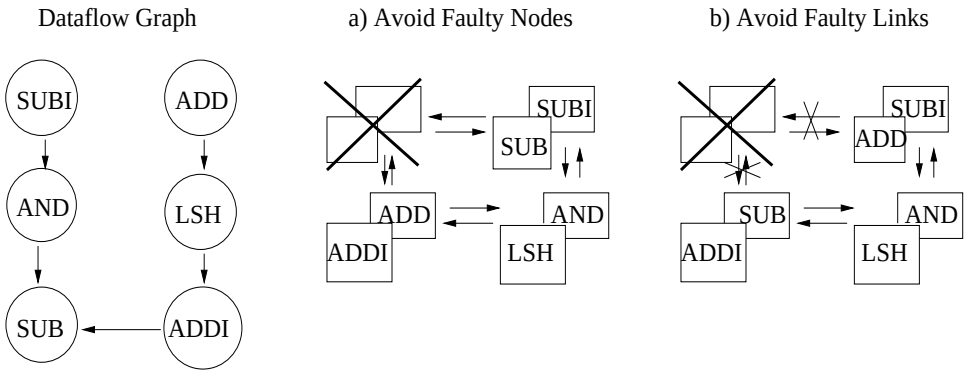

Figure 5.7: Fault-aware instruction placement

struction placement for the same dataflow graph shown in Figure 4.2. With the *faulty node* model, the algorithm must only avoid placing instructions on the faulty ETs. For instance, while `addi` and `sub` in Figure 5.7a are scheduled on functional nodes, the y-x dimension-order path from `addi` to `sub` passes through the defective ET. For the *faulty link* model shown in Figure 5.7b, the algorithm must also ensure that all pairs of dependent instructions can use fully functional ETs along the path specified by the y-x dimension-order routing function. Using the same example, `addi` and `sub` have been rescheduled in Figure 5.7b to provide a functional path between them.

5.3.4 Evaluation Methodology

We used a simulator that models the TRIPS microarchitecture, execution model, and distributed protocols in detail. The simulator was validated to be within 4% of the RTL-level processor simulator on a wide set of microbenchmarks. We employ a variety of benchmarks to model a wide range of behaviors. The benchmarks are compiled using the TRIPS compiler that performs traditional scalar optimizations, and several high-level transformations for generating high quality TRIPS blocks. The TRIPS compiler is still in development and currently lacks a few optimizations that can further improve the quality of the TRIPS blocks. To account for this, the benchmark suite includes both compiler-generated and hand-optimized benchmarks. The hand-optimized benchmarks use better register allo-

Compiler generated EEMBC Benchmarks			
Automotive/Industrial		Networking	
a2time01	Angle to time conversion	ospf	Open shortest path first
basefp01	Basic integer and floating point	pktflow	Packet flow
bitmnp01	Bit manipulation	routelookup	Route lookup
cacheb01	Cache buster	Office Automation	
canrdr01	CAN remote data request	dither01	Dithering
aifirf01	Finite impulse response filter	rotate01	Image rotation
idctrn01	Inverse discrete cosine transform	text01	Text processing
iirflt01	Infinite impulse response filter	bezier02	Bezier curve calculation
pntrch01	Pointer chasing	Telecom	
puwmod01	Pulse width modulation	autocor00	Autocorrelation
rspeed01	Road speed calculation	conven00	Convolutional encoder
tblook01	Table lookup and interpolation	viterb00	Viterbi decoder
ttsprk01	Tooth to spark		
matrix01	Matrix arithmetic		
Hand-optimized EEMBC Benchmarks			
a2time01	Angle to time conversion		
basefp01	Basic integer and floating point math		
rspeed01	Road speed calculation		
tblook01	Table lookup and bilinear interpolation		
bezier02	Bezier curve calculation		
dither01	Dithering		
autocor00	Finite length fixed-point autocorrelation		

Table 5.5: List of EEMBC benchmarks used for evaluation.

cation, predication optimization, and hyperblock formation to improve the overall performance.

Table 5.5 shows the list of compiler-generated and hand-optimized EEMBC benchmarks. We used 24 compiler-generated and 7 hand-optimized EEMBC benchmarks. The EEMBC benchmarks are loop-based benchmarks that execute for a user-defined number of iterations. We compute the number of TRIPS blocks executed for a single iteration, and fast forwarded as many blocks to skip the initialization phase. The number of blocks to be fast forwarded must be recomputed if any of the compilation parameters change. The EEMBC benchmarks were executed for a substantial number of iterations to achieve a reasonable simulation time.

Table 5.6 shows the list of SPEC integer and floating point benchmarks. All the

SPEC CPU Integer	
164.gzip	Compression
181.mcf	Combinatorial optimization
186.crafty	Game playing: chess
197.parser	Word processing
255.vortex	Object-oriented database
300.twolf	Place and route simulator
254.gap	Group theory, interpreter
176.gcc	C programming language compiler
SPEC CPU Floating Point	
168.wupwise	Physics/Quantum chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver: 3D potential field
173.applu	Parabolic/Elliptic differential equations
177.mesa	3-D Graphics library
179.art	Image recognition / neural networks
200.sixtrack	High energy nuclear physics accelerator design
301.apsi	Meteorology: pollutant distribution
188.ammp	Computational chemistry
183.equake	Seismic wave propagation simulation

Table 5.6: List of SPEC benchmarks used for evaluation.

SPEC benchmarks were generated by the compiler. We used 10 benchmarks from the SPEC 2000 floating point (SPEC FP) suite, and 8 benchmarks from the SPEC 2000 integer (SPEC INT) suite. The SPEC benchmarks were simulated for the execution phase determined by using SimPoint [132]. The benchmarks either use the early SimPoint region, or one of the early regions among multiple SimPoint regions if the early region is too far into the program to minimize simulation time. The region is stretched to the boundaries of a non-inlined function call or a return, to ensure that any comparison between different compilations is performed on identical program regions.

5.3.5 TRIPS Block Utilization

Fault-aware scheduling algorithms rely on redundancy to isolate the fault. As illustrated by the simple example in Figure 5.7, the TRIPS block must contain empty instruction slots to allow for the possibility of scheduling its instructions such that the defective ET can be

avoided. To tolerate the single-defective-ET fault model, the algorithm must at least be able to reschedule the instructions on the defective ET to other functional ETs. Since the base scheduling algorithm may schedule up to 8 instructions of the 128 instructions in a block on an ET, the fault-aware scheduling algorithm must find at least 8 available instruction buffer entries in the functional ETs to provide the minimal redundancy for successful fault reconfiguration. This implies that each block can contain a maximum of 120 instructions (= 128 - 8) to have any redundancy at all for the algorithm to exploit. Reducing block utilization can potentially lead to performance loss even in the fault-free case and is the static cost of the technique, as we have to first create redundancy before it can be dynamically exploited when there are defects. This is similar in concept to adding a fixed number of explicit spares for improving availability, which contribute to fixed overheads in area, and possibly execution time. This minimal reduction in block utilization may prove to be inadequate for the more restrictive *Faulty Link* fault model, for now each instruction must not only be scheduled on a functional ET, but also on one that can be reached from all its parents, and from where it can reach all its children through paths that do not contain a defective ET. The *Faulty Link* fault model may require reducing the block utilization even further to improve the robustness of the algorithm.

Table 5.7 presents the results for the average dynamic block size in instructions for the set of compiler-generated EEMBC and SPEC integer and floating point benchmarks, and the hand-optimized EEMBC benchmarks. The results show that although the allowed block capacity is 128 instructions, the average block size of compiler-generated EEMBC, SPEC integer and floating point benchmarks are 45.3, 28.5, and 41.5 instructions respectively. While this is the average dynamic capacity across all the blocks in the benchmarks of that category, individual blocks within benchmarks have block capacities reaching up to 126 instructions. However, the hand-optimized EEMBC benchmarks have a much greater average dynamic block size of 72.5 instructions, demonstrating the effectiveness of the extra optimizations. These optimizations are currently being implemented in the TRIPS compiler

Compiler-generated EEMBC Benchmarks		Hand-optimized EEMBC Benchmarks	
a2time01	45.9	a2time01	85.4
aifirf01	72.5	tblook01	94.2
cacheb01	24.4	bezier02	61.8
canrdr01	30.5	dither01	65.9
tblook01	78.1	autocor00	44.3
matrix01	52.2	rspeed01	51.6
ttsprk01	51.9	basefp01	104.4
pntrch01	79.5	SPEC Integer Benchmarks	
pktflow	36.4	164.gzip	39.3
ospf	36.5	176.gcc	24.1
rotate01	44.6	181.mcf	37.6
bezier02	22.8	186.crafty	29.9
dither01	40.4	197.parser	19.4
text01	33.9	255.vortex	20.2
autocor00	42.5	254.gap	26.7
conven00	42.9	300.twolf	30.8
bitmnp01	58.7	SPEC Floating Point Benchmarks	
rspeed01	30.8	168.wupwise	39.9
puwmod01	37.1	171.swim	36.7
iirflt01	50.3	172.mgrid	74.8
routelookup	39.5	173.applu	45.8
basefp01	43.9	177.mesa	35.9
viterb01	52.6	179.art	48.7
idctrn01	38.7	183.equake	27.1
		188.ammp	26.3
		200.sixtrack	29.9
		301.apsi	44.9
Compiler-generated EEMBC Benchmarks		MEAN: 45.3	
Hand-optimized EEMBC Benchmarks		MEAN: 72.5	
SPEC Integer Benchmarks		MEAN: 28.5	
SPEC Floating Point Benchmarks		MEAN: 41.5	

Table 5.7: Dynamic block capacity of a set of EEMBC, SPEC integer and floating point benchmarks.

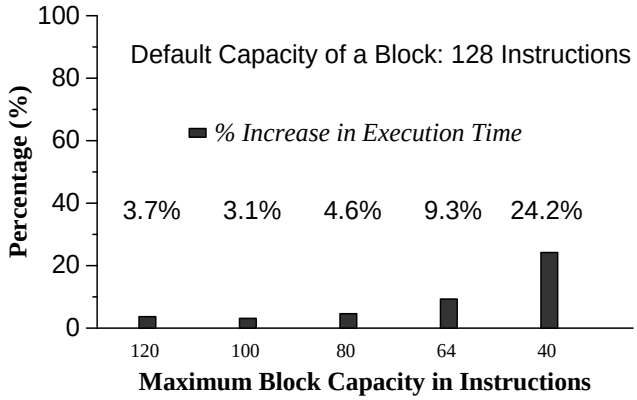

Figure 5.8: Performance sensitivity to block capacity for the set of EEMBC benchmarks

which will reduce the gap between compiler-generated and hand-optimized benchmarks. Overall, these results show that most blocks are not fully utilized and inherently provide the capability required for fault reconfiguration.

Figure 5.8 presents the average results for the performance sensitivity to block capacity for the set of compiler-generated EEMBC benchmarks. The TRIPS compiler can take the maximum allowed block capacity as an input for block generation at static compilation time. The results were generated by specifying these different block capacities as input to the compiler, and measuring the resulting increase in execution time compared to the default maximum capacity of 128 instructions. As described in Section 5.3.4, the number of blocks that have to be fast forwarded is independently determined for each compilation. The results show that execution time increases by less than 5% on the average even if the block capacity is reduced to 80 instructions. On the other hand, reducing the block capacity to 64 or 40 instructions leads to a significant increase in execution time of 9% and 24% respectively.

Overall, these exploratory results show that most of the blocks already have sufficient redundancy in them, and reducing the maximum block capacity to as low as 100 instructions to enforce redundancy in all the blocks affects performance by less than 4%. Since fault-aware scheduling algorithms are predicated on the capability to regulate the

block size to allow fault reconfiguration, we restrict our evaluation of fault-aware instruction placement to the compiler-generated benchmarks. While the block size of hand-optimized benchmarks are fixed, the TRIPS compiler, as described above, allows the the maximum block capacity to be specified as an input at static compilation time.

5.3.6 Fault-Aware Instruction Placement

The basic idea of fault-aware instruction placement is to expose the defective configuration to the scheduler, so that it can appropriately perform instruction scheduling based on the configuration and the fault model. While this concept obviously applies during static compilation time to improve yield, it can also be extended to improve lifetime reliability. Section 5.3.7 discusses the challenges associated with run-time compiler-assisted reconfiguration. Section 4.2.2 described the base scheduler for the TRIPS architecture that takes as input the instructions, and a detailed processor model that includes the routing topology, static instruction and communication latencies, and produces the instruction schedule containing the assignment of instructions to ETs. We describe two heuristics for fault-aware instruction placement built on the base scheduling algorithm.

Avoid Faulty Nodes Heuristic (AFN)

This heuristic assumes the *Faulty Node* fault model. The scheduler takes as input a detailed processor configuration which additionally includes pointers to the defective ETs, the rest of the inputs are identical to the base scheduler. The algorithm itself is identical to the base scheduler, but it now considers only the functional ETs for instruction placement. Based on the discussion in Section 5.3.3 for the single-defective-ET model, setting the block utilization at 120 instructions ensures the success of this heuristic. Since there is only one defective ET (and no faulty links), there is always enough redundancy to reschedule the instructions when the block is constrained to have a maximum of 120 instructions at static compilation time. To limit the simulation time, we measured the performance impact for

a subset of the 16 possible fault locations, one in the first row next to the RTs, one in the first column next to the DTs, and the third in the middle of the ET array. We measured the average performance impact to be less than 4% over the set of compiler-generated EEMBC, and SPEC benchmarks. Based on this result, and the performance sensitivity to block capacity measured in Section 5.3.5 (see Figure 5.8), we conclude that the performance drop is mostly due to the reduction in block capacity, and rescheduling the instructions to avoid the defective ET has relatively minimal impact for these set of benchmarks.

Avoid Faulty Links Heuristic (AFL)

This heuristic accounts for both faulty ETs and routing paths, and can be considered as an enhancement to the previous algorithm. The input processor configuration now not only contains pointers to the defective ETs, but also has infinite communication latencies assigned to the defective nodes from all of its immediate neighbors. This implies that any path from a producer to a consumer instruction that includes a defective execution node is of infinite duration, which naturally serves to ensure that such a path is never selected to provide the block with the minimum completion time. However, the constraints imposed by faulty communication paths and dimension-order routing can potentially lead to intermediate block schedule configurations from which it is impossible to find a legal and functional location for the next dependent or parent instruction. Figure 5.9.a illustrates an example placement of the parent instructions (P1, P2), the faulty ET, and the ETs with one available instruction buffer entry (A). In this example, there is no ET with available entries where the child can be placed that gives functional routes (that follow y-x dimension-order routing) from both the parents. Figure 5.10.a presents another example placement of the parent instructions (P1, P2) and the faulty ET, for which there is no ET, regardless of whether it has available entries or not, where the child can be placed that gives functional routes (that follow y-x dimension-order routing) from both the parents.

The key idea the AFL algorithm uses is to backtrack the existing schedule by

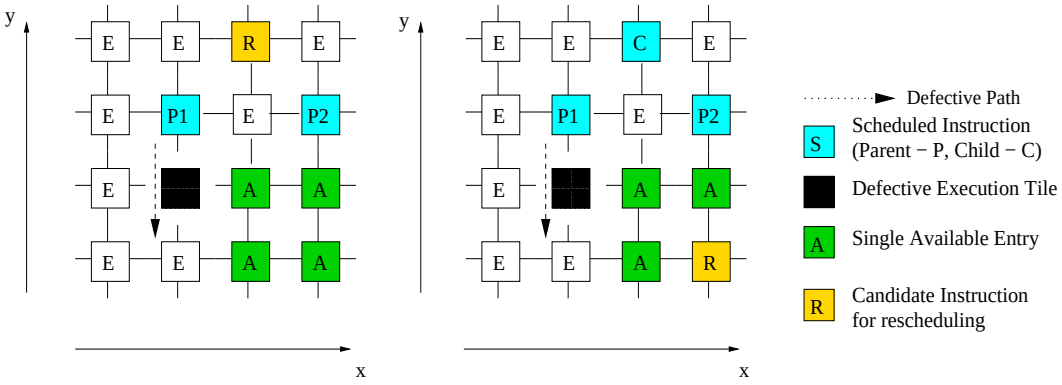

Figure 5.9: Fault reconfiguration by rescheduling a pre-placed instruction, to create a new location for the current instruction which provides it with functional routes from both its parents

rescheduling some of the instructions that have already been placed. The algorithm searches for all locations, regardless of whether they already contain instructions or not, that provide the current instruction functional paths to all its pre-scheduled parent and child instructions. If such a location contains a pre-placed instruction, the algorithm attempts to reschedule the instruction to an alternate valid location, and schedules the current instruction in the entry freed by this relocated instruction. However, rescheduling one instruction may trigger rescheduling a dependent instruction to provide a functional path between them, and lead to a domino effect that does not terminate until the algorithm is able to find functional locations and paths between all the instructions in the affected data dependence graph. To limit the impact on scheduling latency and complexity, and to tune the algorithm for the single-defective-ET fault model, we constrain the depth of rescheduling and limit it to rescheduling only the current instruction or the immediate dependent parent or child instructions.

For instance, to fix the unfavorable block schedule configuration shown in Figure 5.9.a, the algorithm reschedules a pre-scheduled instruction R to a different legal location that is functional also, and places the current instruction C in the location freed by

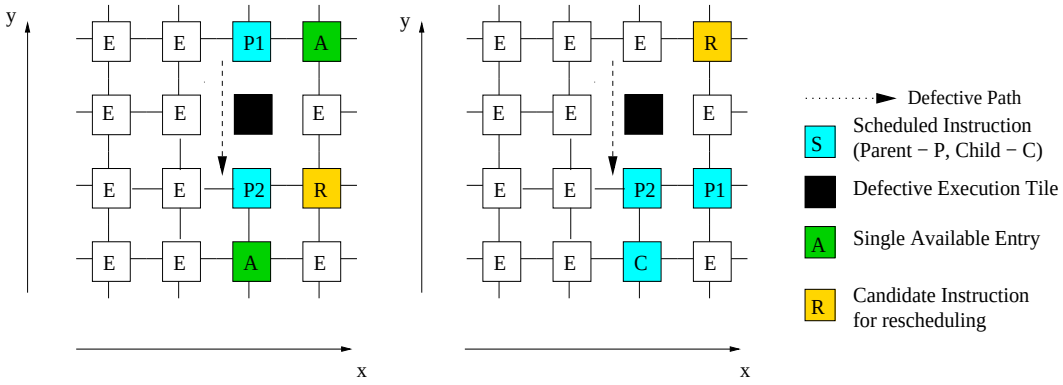

Figure 5.10: Rescheduling a pre-placed parent instruction changes the configuration such that it allow the current child instruction to be successfully scheduled

R as shown in Figure 5.9.b. R can only be relocated to an ET with a currently available entry (A), and is not allowed to reschedule another dependent instruction. However, this solution will not work for the situation in Figure 5.10.a since there are no legal nodes which will satisfy C. Hence, in this case the AFL algorithm reschedules one of the pre-scheduled parent instructions $P1$. The algorithm first attempts to find an ET with an available entry (A) that $P1$ can be relocated to. Since there are no such ETs in this example that provides functional paths from both parents, it reschedules another instruction R, as shown in Figure 5.10.b, similar to the previous scenario. In general, as many pre-scheduled immediate dependent parent or child instructions are rescheduled in order to successfully schedule the current instruction C. The AFL algorithm fails if it is unsuccessful in scheduling C using either of these two approaches.

Figures 5.11.a and 5.11.b show two more example locations for the defective ET, that expose the limitations of the AFL algorithm with respect to anchor points. The first example shows how the location of the defective ET, immediately next to an RT on the periphery of the execution array, makes it impossible to access the required architectural register from the register bank using y-x dimension-order routing. In this study, we make

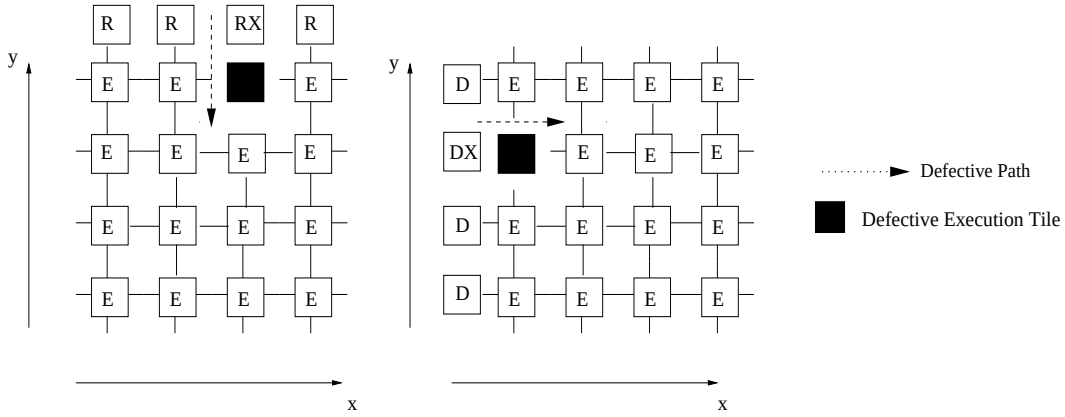

Figure 5.11: Defective execution tiles at the register tile and data tile boundary can make it impossible to access a particular register or data cache bank. Dedicated bypass logic, software support, or programmable hardware support is required to handle these defects.

the simplifying assumption that the hardware must include redundant logic and communication paths to deliver register operands bypassing the defective ETs on the RT boundary, while defective ETs from the second row onwards can be handled by the AFL algorithm described above. Alternatively, as described earlier, fault-aware register allocation can solve this problem by not using the registers from this RT in the program. However, while instruction scheduling is localized to a single block, register allocation is a global operation in the compiler and hence is much costlier.

The example in Figure 5.11.b shows a defective ET that makes it impossible to access a particular data cache bank. Again, we make the simplifying assumption that the hardware must include redundant logic to deliver load and store operands bypassing the defective ETs on the DT boundary, while defective ETs from the second column onwards can be handled by the AFL algorithm described above. Load and store accesses are different from register accesses, because the exact cache bank that needs to be accessed is not known until dynamic execution time, since it depends on the actual address. As described earlier, an alternative to including dedicated bypass logic, would require programmable hardware

support for modifying the address interleaving policy across the data cache banks so that no addresses are mapped to the cache bank adjacent the defective ET.

Results: Based on the discussion in Section 5.3.5, we explored two block sizes, 120 instructions, and 112 instructions with more than the minimal redundancy. First, we studied whether the AFN heuristic used for the Faulty Node model is able to tolerate both faulty nodes and links. We performed instruction scheduling for all the benchmarks for all the 16 possible fault locations, and found that the AFN heuristic failed to find a functional schedule in over 75% of the cases. In the remaining cases, the algorithm was luckily able to find a schedule that provided functional nodes and paths for all the block instructions.

On the other hand, the AFL algorithm is successful in scheduling all the benchmarks for all the 16 fault locations. Further, we found that the AFL heuristic was equally successful for blocks with 120 instructions and blocks with 112 instructions, proving the extra block redundancy as unnecessary in this context, and demonstrating the effectiveness of the algorithm. However, reducing the block utilization indirectly increases the robustness of the algorithm, and hence may allow the designer to reduce its complexity. Section 5.3.8 expands on this tradeoff. Finally, we measured the average performance impact to be less than 4%, similar to the AFN algorithm, for the same subset of fault locations. This result again suggests that the performance drop is mostly due to the reduction in block capacity, and rescheduling the instructions to avoid the defective ET and network links has relatively minimal impact. We recognize that these results are dependent on the compiler, and are worth re-investigating as the TRIPS compiler matures.

Overall, the results show that fault-aware instruction scheduling has potential to increase the yield at a small performance overhead, and hence provide a high Y_{PAV}.

5.3.7 Compiler-Assisted Fault Reconfiguration for Improving Lifetime Reliability

This section provides a brief overview of how fault-aware instruction placement can be extended to improve lifetime reliability. Runtime BIST mechanisms must periodically perform fine-grained error detection, and expose the defective configuration to the scheduler [24]. On any change in the defective ET configuration, program execution has to resume from a known good checkpoint since the execution on the defective processor cannot be trusted [24]. Resuming from the checkpoint, the instruction scheduler must use the new defective configuration to perform fault-aware instruction scheduling. When the fault-aware scheduling heuristic is successful on a portion of the program, it can be executed on the degraded processor. If it is unsuccessful, program execution must be shifted to a fully functional processor. Applications can use a simple policy wherein a migration marks the specific processor as unusable by it in the future, and the program is executed on a fully functional processor until completion. Therefore, processors are marked defective on a per-application basis, providing an application-specific defective configuration that potentially improves performability [136]. This is a fine-grained compiler-assisted adaptation of dynamic processor sparing implemented in IBM systems, where an application is transparently migrated from a defective processor to a functional processor and resumes execution from a checkpoint [28].

5.3.8 Tradeoff between Block Utilization and Algorithm Complexity

Good design of the fault-aware scheduling algorithm improves chip yield in the static case. In the dynamic case, it reduces the probability of application migration and increases both the availability and the performability of the system [136]. The two key design factors are the amount of redundancy in the block or the block utilization, and the complexity of the scheduling algorithm. While decreasing the maximum allowed block capacity even further increases the amount of redundancy and indirectly improves the robustness of the algorithm,

the disadvantage is that it is a one-time static decision that contributes to fixed performance overhead even in the fault-free case (see Section 5.3.5). Alternatively, the complexity of the scheduling algorithm can be increased to tolerate larger number of defects, or more difficult defective configurations. Ideally, the block utilization and the algorithm must be designed for a single-defective-ET fault model in the beginning, and upgrades to the algorithm can be made over time if necessary. This is similar in concept to updates to the code morphing software in the Transmeta processor, to adapt to modifications in the VLIW architecture for achieving higher performance.

5.3.9 Hybrid Hardware-Software Approach for Fault Reconfiguration

While we propose a compiler-assisted fault reconfiguration technique, a large body of research exists on adaptive fault-tolerant routing for n-dimensional meshes [120, 137, 137, 138]. A purely hardware based scheme using adaptive fault-tolerant routing protocols would still require explicit spare ETs to accommodate the instructions that are scheduled on the defective ET in the single-defective-ET fault model. Further, based on the location of the spare ET, extra reconfiguration logic will be needed to redirect the block instructions on the GDN (instruction dispatch network) to the spare ET, and redirect instruction execution results originally targeted for the defective ET to the spare ET. To eliminate the addition of explicit spares since the architecture already contains abundant redundancy, and to reduce the complexity of the reconfiguration logic, we envision a more efficient, hybrid approach for fault reconfiguration using both the compiler and the adaptive routing protocols in co-operation.

In the hybrid approach, the compiler can reduce the maximum block capacity to ensure that the functional ETs have enough capacity to accommodate the entire block, thus obviating the need for spare ETs. Further, the scheduler can use the simple AFN heuristic described in Section 5.3.6 to avoid placing instructions on defective ETs, guaranteeing that no instruction result targets a defective ET, thus getting rid of this extra reconfiguration

logic. The software assistance reduces the responsibility on the hardware, and the adaptive routing algorithms are now only responsible for correctly delivering packets from source to destination ETs, routing around the defective ETs within the 4x4 array. The routing algorithms typically use virtual channels to route around defective nodes, and follow certain conventions with respect to virtual channel ordering for achieving deadlock freedom [120, 138].

While virtual channels can provide higher throughput and fault tolerance, they also incur fixed overheads in buffer space, and latency for virtual channel management. Gratz et al. showed that the latency of the operand network (OPN), that is tightly integrated into the execution pipeline in the ET, is critical for performance [110]. Further, they also conclude that the OPN router buffers occupy significant area, and the area cost must be carefully balanced with the performance benefits. Hence, while adaptive routing provides good theoretical guarantees for fault tolerance, more analysis is required to optimize the latency, and area tradeoffs, and we believe it is a promising avenue for future work.

5.4 Related Work

Support for Defect Detection: Regardless of the source of yield loss that is targeted, yield enhancement schemes rely on the ability to detect faults in the circuit, and whenever possible, circumvent the problem by disabling or reconfiguring the faulty resource [41]. As the number of transistors on a chip increases, so does the complexity and volume of the test patterns required to identify and diagnose faults. Driven by the enormous expense of suitable testers and test time with growing chip complexity, many chips today are augmented with built-in self test (BIST) functions to partially relieve the testing burden. Although BIST does not eliminate the need for traditional test patterns, it is used extensively for testing on-chip RAM structures, for stressing the design during burn-in, and for speed binning. Some RAM BIST controllers are capable of pinpointing faults within the array, configuring the redundant rows or columns to avoid the fault, and then retesting the array. If successful,

the BIST controller communicates the recommended configuration information to the next level of test coordination.

In this study, we propose to push the traditional techniques of yield enhancement, based on detecting and either disabling or reconfiguring the faulty resources [41], inside the boundaries of a single processor by identifying and exploiting redundancies at the microarchitectural level. Due to the fine-grained nature of redundancy proposed, we envision the need for more advanced BIST controllers that build on the capability that exists for array repair to include support for other types of fault tolerance mechanisms [23]. The challenges here will be to define the appropriate granularity for the BIST domains, and to develop automatic pattern generators for isolating faults in structures that contain more logic circuitry than basic RAMs. Shyam et al. showed that such advanced BIST techniques are possible and can be integrated into a processor with modest modifications to the hardware [24]. Schuchman et al. proposed *intra-cycle logic independence* (ICI) as the necessary condition for testability to allow fine-grained fault isolation in processor structures, and outlined modifications required to conventional processor structures to comply with ICI [139]. Another promising concept for chip multiprocessors (CMPs) proposed in [140] involves the use of one of the on-chip processors themselves for coordinating the testing of other processors and chip-level structures.

Yield Metrics: There are two classes of work related to the *performance-averaged yield* concept. In [141], yield evaluation is done for memory chips with redundancy that allows the chip to be partitioned so that the fault-free sections can operate independently. The *equivalent yield* concept proposed in that paper accounts for partially good chips by scaling the yield by the memory capacity of the degraded chip. In this study, we extend the argument to processor chips and propose a performance based metric that is a better measure of the effect of chip degradation at the system level. Performability was proposed as a refined measure of availability by accounting for the degraded performance dynamically [136]. *Performance-averaged yield* adapts this dynamic concept to static chip yield evaluation by

accounting for the relative performance of the degraded chips.

Design for Yield: Section 2.1.2 described several techniques for improving defect tolerance at the device and circuit level. Bower et al. proposed self-healing arrays that detect array defects at run-time and dynamically perform reconfiguration [142]. DeHon et al. proposed seven strategies based on redundancy at different granularities, support for roll-back recovery, and run-time reconfiguration support for tolerating high defect rates expected in future architectures with nanoscale devices [143].

Design for Lifetime Reliability: Several designs based on coarse-grained spatial redundancy have been proposed to improve the lifetime reliability of systems [18, 25, 28]. Recent research has focused on applying similar concepts in the context of future multi-core or system-on-a-chip architectures. Aggarwal et al. identified simple modifications that can be made to a commodity multi-core processor to enhance its capability for fault isolation and thus prevent the entire chip from failing on a defect [26]. Sylvester et al. proposed a complete architecture called ElastIC based on aggressive run-time self-diagnosis, adaptivity, and self-healing to tolerate parametric defects due to process variability [144]. Srinivasan et al. extended the concept of *performance-averaged yield* to improve the availability of a processor by allowing it to continue operation in a degraded state [145]. Srinivasan et al. also proposed dynamic reliability management, a fault evasive technique for probabilistically improving processor lifetime [31].

Software-assisted Hard Error Reliability: Many high-end systems use a combination of software, firmware, and hardware to both detect and manage errors [15, 28]. IBM pSeries systems use self-diagnosing software to monitor the recoverable error rates of processors [28]. If the number of errors exceed an internal threshold, the operating system is notified which in turn triggers *dynamic reconfiguration* to substitute the defective processor. Recently, Joseph et al. proposed a virtualization based methodology to dynamically

spare an entire defective processor in a CMP [126]. Sankaralingam et al. argue that future tile-based CMP designs using larger cores will be able to achieve technology scalable high performance over a wider range of applications [111]. Consequently, there will be fewer processors per chip, which motivate the hardware and software techniques for exploiting intra-processor redundancy proposed in this chapter.

5.5 Summary

This chapter examines the redundancy in modern microarchitectures that can be used to enhance their yield. Though we focus primarily on the yield loss due to random defects, we recognize that many of the techniques discussed here will also help in identifying more usable chips during the initial technology learning phase, and in improving the lifetime reliability of the processor. We propose a new yield metric called *performance-averaged yield* (Y_{PAV}) which accounts for the level of performance degradation on all functioning chips. For dynamic superscalar architectures, we focus on relatively coarse grain components within the microarchitecture such as execution units and cache banks, and show that mechanisms already exist within the processor control logic that can easily disable the defective components from being used during program execution.

The results also demonstrate that while using just inter-processor redundancy achieves high yield, exploiting microarchitectural redundancy within a processor further improves the yield, and will be increasingly important if future CMPs use coarse-grained cores to achieve high performance. By exploiting microarchitectural redundancy, we demonstrate that the Y_{PAV} can be improved to as high as 99.6% at 50nm, with a maximum reduction in performance in any chip of less than 20%, a substantial improvement from a $Y_{OVERALL}$ of 60% achieved when only considering the defect-free parts. While AR, DQR, and CLR are applicable in static architectures also, we argue that software-assisted fault reconfiguration may be more efficient than pure hardware based mechanisms for statically allocated resources on a distributed execution substrate. We evaluate fault-aware scheduling heuristics

in the TRIPS architecture which, for the set of benchmarks explored, successfully reschedule the program to avoid defective ETs with less than 4% loss in performance.

Today's systems that provide fail-in-place capabilities do so at the system level and typically provide hot spares for power supplies, processors chips, memory modules, and disks [16]. We advocate pushing fail-in-place inside the boundaries of a single chip or processor and allowing defective components to continue to operate, perhaps with somewhat degraded performance. Optional components arise in system-on-a-chip (SOC) implementations in which a subsystem of the chip could be defective, but the chip could be deployed in systems that did not require the functionality of the defective component. Extending performance binning further, *functional binning* is a similar strategy that recognizes some design features to be *optional* from a yield binning point of view. For example, future systems-on-chip (SoC) designs may include a wide range of modular functions (GPS radio, WiFi radio, etc) that are not essential for all market applications but are included to enable a single product to achieve higher volumes. Allowing some of these features to be optional enables functional binning, and can have the effect of increasing the effective yield.

The regularity and redundancy that we exploit is synergistic with several technology and design trends. Managing increasing design complexity has become a tremendous challenge for both design and verification, and demands modular design techniques that reuse chip components, thus creating redundancy opportunities. Second, the increase in wire delay relative to transistor switching time will likely lead to partitioned architectures composed of replicated hardware modules [111]. whether they be processors, ALUs, or a combination of both [50]. Finally, looming limits on energy and heat have led architects to suggest trading power for performance by modulating the clock frequency in different regions of a chip [146], dynamically reducing memory structure sizes, or by selectively disabling microarchitecture components [147]. We expect that future systems designers will take advantage of replication and partitioning to meet these joint goals of power, performance, reliability, and ease of design.

Chapter 6

Soft Error Reliability Regimes

Different systems have different reliability requirements based on their target application and market domain. For instance, high availability is a critical requirement for enterprise systems used in business processes [148]. On the other hand, processors used on desktops have less critical reliability requirements, and companies typically have different soft error rate targets for these different market segments. The relative importance of reliability determines the degree of overhead that is acceptable due to reliability enhancement techniques, and is an explicit design parameter when designing these systems. The masking factor or the fraction of soft errors that have the potential to affect program outcome, and the reliability target together determine the degree of protection needed. Therefore, an accurate estimation of the masking factors at all levels is important to prevent over or under-designing the reliability mechanism.

Chapter 3 described the various factors that determine whether a soft error occurs in an SRAM cell, or a latch, either because of a direct strike to the latch or due to propagation through combinational logic. Figure 6.1 illustrates the classification proposed by Mukherjee et al. of the possible outcomes a single bit fault at the architectural level, once the soft error has already occurred [2]. In three cases, 1, 2, and 3 the soft error has absolutely no effect on the program. In scenarios 1 and 3, the soft error is masked at the microarchitec-

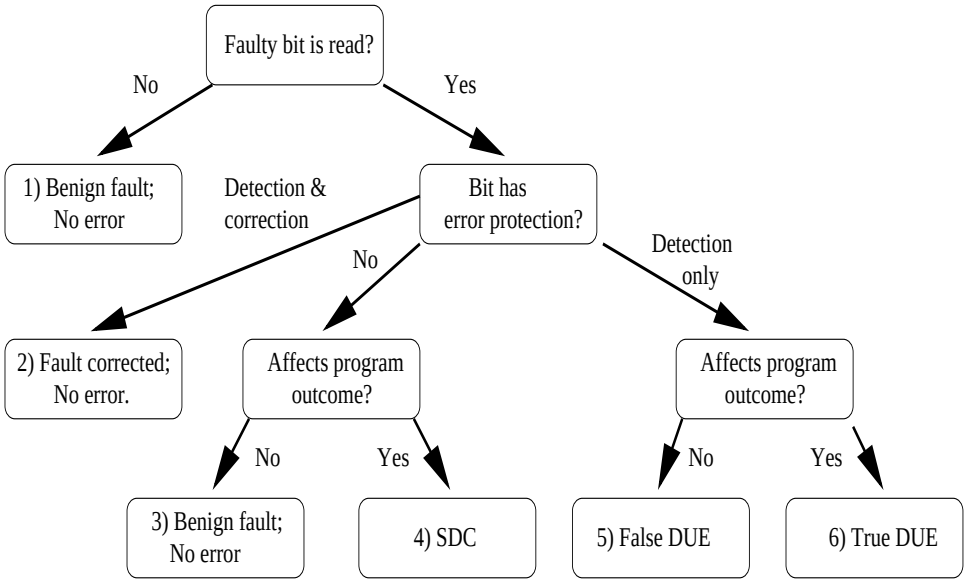

Figure 6.1: Mukherjee et al. proposed this classification of a soft error in a bit based on its severity [2]. SDC = silent data corruption, DUE = detected unrecoverable error.

tural or architectural level, analogous to circuit-level error masking discussed in Chapter 3. In this chapter, we improve the accuracy of chip-level SER estimation by extending the methodology in Chapter 3 to account for microarchitectural and architectural masking factors. While the error is detected and corrected in scenario 2, there is no provision for error correction in scenarios 5 and 6. Adding detection alone to a bit prevents the program from generating a wrong output, but provides only fail-stop behavior. This class of errors is referred to as detected unrecoverable errors (DUE). The worst scenario is 4, where the bit has no protection and silently affects the program output. This is referred to as silent data corruption (SDC), and designers employ error detection and correction mechanisms to reduce the probability of SDC. The reliability targets for a particular design are specified in terms of the acceptable SDC and DUE rates. For example, IBM targets 114 SDC FIT, 4566 system-kill DUE FIT, and 11415 process-kill DUE FIT for its Power4 systems [2]. Process-kill DUEs correspond to detected unrecoverable errors which can be isolated to a specific

process or set of processes that have to be killed. On the other hand, the entire system must be restarted to recover from a system-kill DUE. As expected, the SDC target is more stringent than the system-kill DUE target, which is more stringent than the process-kill DUE target.

This chapter provides the foundation for the architectural soft error reliability techniques we design and evaluate in this dissertation. This chapter makes the following main contributions:

Understanding the scope of architectural reliability techniques: We propose a systematic methodology for quantifying the reliability improvement required every technology generation from architectural techniques. We present four different scenarios for architectural reliability improvement depending on the support for improving reliability at other levels of the design. Since the reliability requirement is significantly different in the four scenarios, we argue that different architectural reliability enhancement strategies should be adopted to provide the best performance-reliability tradeoff.

Reliability performance ratio: We propose a new metric for quantifying the performance-reliability tradeoff that can be applied in all the scenarios. The metric allows a designer to account for both the performance overhead and the reliability improvement, and choose the technique that provides the best tradeoff.

Soft error reliability regimes: Building on this framework, we classify architectural soft error reliability techniques into four regimes: error correcting codes, full redundant execution, selective redundant execution, and AVF throttling, appropriate for the four different scenarios. While architectural techniques that fall within the first three regimes have been proposed in prior research, we present a new regime called *AVF throttling* that can significantly reduce the power consumption, and complexity overhead. This classification sets a good starting point for the specific reliability techniques from each regime we evaluate in

the TRIPS architecture in Chapter 7.

Architectural masking factors in the TRIPS architecture: We present a systematic, detailed methodology and evaluation of the architectural masking factors of important structures in the distributed TRIPS architecture. We compare the architectural masking factors in the TRIPS architecture with conventional architectures, and identify the causes that lead to the differences in soft error vulnerability. We conclude with some important implications of physical and logical partitioning, employed in distributed architectures, on soft error vulnerability.

Section 6.1 builds a framework for estimating the reliability improvement needed every generation from architectural techniques. Section 6.2 describes the ACE analysis methodology for estimating architectural masking factors, and Section 6.3 presents our new metric RPR for evaluating the performance-reliability tradeoff. Section 6.4 divides architectural soft error reliability mechanisms into four regimes with different performance-reliability tradeoffs. Section 6.5 summarizes the masking factors for important structures in the TRIPS architecture. Section 6.6 discusses the tradeoffs between the two primary methods for evaluating architectural masking factors. Section 6.7 presents our conclusions and builds the foundation for evaluating each regime in detail.

6.1 Soft Error Rate Scaling Analysis

Chapter 3 projected the raw soft error rates for the three basic circuits of a modern microprocessor: SRAM cells, latches and combinational logic. Several scaling studies basically agree that the raw soft error rate(SER)-per-SRAM cell will decrease gradually with decreasing device size, although they differ on the rate of decrease [67, 149]. Karnik et al. conservatively project SER-per-latch to either increase by 8% every generation if supply voltage scales linearly (0.7x) across technologies, or remain constant if voltage scaling slows down, and supply voltage decreases only by 0.8x every generation [88]. However, they also con-

clude that even with slowing voltage scaling the SER-per-latch can increase by up to 20% every generation depending on the device and charge collection parameters. Our results in Chapter 3 show a slight increase in SER-per-latch every generation. In Chapter 3, we predict that the raw SER of combinational logic will be comparable to that of latches by the 50nm technology generation [67]. However, we don't consider the impact of circuit-level logical masking which can potentially reduce the observed rate of increase [70, 71].

Building on these raw error rates for the building blocks, Chapter 3 also projected the chip-level SER to account for the increase in the number of bits every generation. It used a simple chip area model based on Moore's law, and a simple analytical model to divide the chip area among SRAM cells, latches, and combinational logic. While this simple model accounts for the difference in raw vulnerability of the three basic circuits, it treats all microprocessor structures built using the same basic circuit as equally vulnerable. The following equation shows the simple model for chip-level SER used in Chapter 3:

$$\text{FIT/chip} \quad = \sum_{SRAM,latch,logic} FIT/bit \times Bits$$

where FIT/bit represents the raw SER of each circuit type, and in this equation also accounts for the circuit-level masking factors described in Chapter 3, and Bits stands for the total number of transistors alloted to the particular circuit type. The model accounts for circuit-level masking factors, but it does not account for masking at the microarchitectural and architectural level which can further reduce the error rate. For instance, while both an on-chip cache and a branch predictor use SRAM cells for their storage, an error in the branch predictor state will only affect performance, whereas an error in the cache can lead to faulty program output. In summary, the raw soft error rate (taking into account the circuit-level masking factors), the structure capacity, and several architectural and microarchitectural masking factors together determine the soft error vulnerability of the structure. The following equation illustrates the refined model for chip SER:

$$\text{FIT/chip} = \sum_{processor structures} FIT/bit \times AVF \times Bits$$

where the additional term, AVF, stands for *architectural vulnerability factor* per bit, and represents the architectural and microarchitectural masking factors. It is a measure of the likelihood of an error propagating through that bit to the program outcome, given that the error was not filtered by the circuit-level masking factors. Chip SER is determined by combining three factors: the SER-per-circuit, the total number of bits, and the architectural vulnerability factor (AVF). The goal of a reliability mechanism is to maintain the system SER, by reducing these three factors using different design techniques. An accurate estimation of the masking factors at all levels is important for precisely determining both the absolute error rate, and the scaling trend across technologies, and is necessary to prevent over- or under-designing the reliability mechanism. While device or circuit level techniques can be used to reduce the raw SER/bit, error correcting codes have been traditionally used to reduce the total number of vulnerable bits in a design. Processor state that cannot be effectively protected using these two techniques must be protected using microarchitectural reliability mechanisms that decrease the AVF of the structures to achieve the required reduction in error rate.

Figure 6.2 shows four different scenarios for maintaining chip SER, with different assumptions for the error rate per bit (SER/Bit), and the number of vulnerable bits (Bits). Each of the four scenarios places a different requirement on the required reduction in AVF, and thus implies a different performance-reliability tradeoff at the microarchitectural level. From a purely technology scaling viewpoint, the first scenario shows that nearly 51% reduction in AVF would be required every generation, if the number of bits follows Moore's law scaling and almost doubles every generation, and the (SER/Bit) increases by 8% [67, 88, 149]. Karnik et al. demonstrated that 70% of the hardware logic paths are

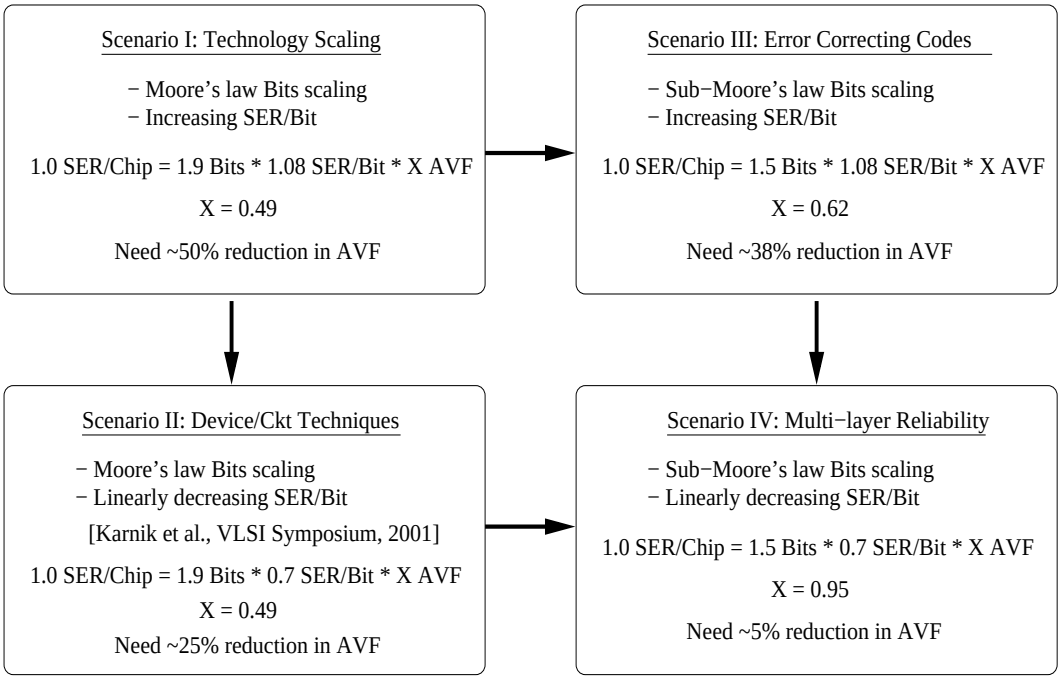

Figure 6.2: Soft error rate scaling analysis

non-critical with respect to clock frequency, and exploited this slack to perform selective capacitance insertion into the latches at the end of these paths to achieve an almost 2X reduction in SER, and a linearly decreasing SER/Bit [11, 88]. Scenario II shows that such device or circuit techniques can reduce the AVF reduction needed to 25%. Techniques such as ECC or parity will continue to be used in the future to protect an increasing fraction of on-chip storage structures. While Moore's law implies a 100% increase in processor state every generation, these techniques help in achieving a slower than Moore's law rate of increase in vulnerable processor state. Assuming they are able to decrease the rate of increase of vulnerable bits every generation to 50%, the AVF reduction required then drops to 38%, as shown in Scenario III. Finally, if both device and circuit techniques, and error correcting codes, are used to reduce both the (SER/bit), and the rate of increase of vulnerable bits (Bits), the reduction needed in AVF is only 5% every generation.

Since the reliability requirement is significantly different in the four scenarios, we

argue that different architectural reliability enhancement strategies should be adopted to provide the best performance-reliability tradeoff. A designer can use this characterization to choose the appropriate architectural reliability technique based on his reliability requirement and acceptable design overhead. Circuit level techniques and error correcting codes also have associated implementation costs. Karnik et al. showed that the hardening techniques they propose have a 10% cost in setup time, and a 4% penalty in power consumption [88]. Error correcting codes also typically need dedicated circuitry and have a fixed storage overhead [21]. Mitra et al. present a complete characterization of circuit-level soft error reliability techniques based on their reliability improvement, and area, power, and complexity overheads [150]. In the same spirit, we classify architectural soft error reliability techniques into four regimes that map to these different scenarios in the AVF reduction space, and different points on the performance-reliability tradeoff spectrum. Before we present this classification, the next two sections describe the methodology used to compute the AVF, and the metric for quantifying the performance-reliability tradeoff of architectural reliability techniques.

6.2 Computing the Architectural Vulnerability Factor

Mukherjee et al. proposed an innovative methodology based on detailed microarchitecture simulation termed, ACE analysis, to determine the AVF [123]. The AVF of a structure in this technique is measured by tracking each physical bit location of the structure over the period of simulation, and computing the fraction of time for which a soft error in that bit location will affect program output. A bit in which a soft error will lead to incorrect program output, is required for *architecturally correct execution* (ACE), and is termed as an ACE bit. On the other hand, bits in which soft errors are masked are called un-ACE bits. The methodology works by accumulating for each physical bit location the number of cycles for which it is ACE (ACE Cycles), and the number of cycles for which it is un-ACE (un-ACE Cycles). The key insight of the algorithm is that there are several reasons why a

bit can be un-ACE. Mukherjee et al identified several sources of un-ACE bits:

- Invalid state: When a bit in a structure does not contain any valid information it cannot affect the program outcome. For instance, a soft error in an empty slot of the instruction window that does not a contain valid instruction is irrelevant to the program outcome.

- Mis-speculated state: Bits that correspond to mis-speculated state either due to branch mispredictions or incorrect load-store disambiguation are un-ACE.

- Predictor state: Errors in predictor structures will in most cases only affect performance and not functional correctness of the program.

- Ex-ACE state: This is slightly different from invalid state, and corresponds to state that may still be marked valid but has already been utilized by downstream logic. For instance, an instruction that has already dispatched for the last time may still be marked valid, but is not vulnerable since the information is not needed any more.

- NOP instructions: Most of the bits in a NOP instruction, except the bits that differentiate it from a non-NOP, are un-ACE. This is typically the opcode bits of the instruction, but can also be the destination register.

- Performance enhancing instructions: Except the opcode bits, the remaining bits of a performance enhancing instruction are un-ACE. For instance, an error in a prefetch instruction can cause it to fetch data from a wrong location potentially reducing the performance improvement but with no effect on functional correctness.

- Predicated-false instructions: Predicated instructions execute only if they receive enabling predicates during execution. An instruction is indirectly predicated-false if its results are only delivered to a predicated-false instruction. In either case, the majority of instruction and result bits are un-ACE. The predicate specifier bits in the opcode,

the predicate register specifier, and the least significant bit of the predicate value are ACE.

- Dynamically dead instructions: Program execution can also contain dynamically dead instructions whose results are not used, or are used by instructions which are in-turn dynamically dead. In both these cases, a significant fraction of the instruction and operand bits are un-ACE. The opcode and the register specifier bits are still counted as ACE since they can either cause the program to crash during instruction decode, or cause the instruction to deliver the results to incorrect destination registers and thus corrupt program state.

- Logical masking: Logical masking arises mostly due to bitwise logical operations, and compare instructions before branches. The operand bits that are logically masked are un-ACE bits. More subtle scenarios of logical masking may exist and more sophisticated analysis will be required to reveal those opportunities.

The AVF of the bit is then equal to the fraction of the simulation cycles that are ACE Cycles. Assuming that all cells have equal raw soft error rates, the AVF of a structure is the average AVF of all its constituent bits. The following equations show how the soft error rate of the chip (FIT/chip) can be calculated as a function of ACE cycles.

$$\begin{aligned} \text{FIT/chip} &= FIT/bit \times AVF \times Bits \\ &\propto \frac{(AVF \times Bits \times Cycles)}{Cycles} \\ \text{Error Rate} &\propto \frac{ACE\ Cycles}{Cycles} \\ \text{Number of Errors} &\propto ACE\ Cycles \end{aligned}$$

where FIT/bit stands for failures in time per bit, AVF stands for *architectural vulnerability factor* per bit which is a measure of the likelihood of an error propagating through

that bit, Bits stands for the total number of bits in the processor vulnerable to errors, Cycles is a measure of the time for which we are observing the processor for errors, and ACE cycles is a measure of processor reliability, and stands for the cycles that contribute to *architecturally correct execution*. *ACE cycles* is computed by adding up the cycles vulnerable to soft errors, for all the bits, across all the processor structures. The total number of errors is thus directly proportional to the *ACE cycles* accumulated during that period.

6.3 Reliability Performance Ratio (RPR)

The equations show that reducing the error rate requires a corresponding reduction in ($ACE\ Cycles/Cycles$) by the same amount. An ideal reliability technique would achieve this by only decreasing the $ACE\ Cycles$ without affecting $Cycles$, thereby reducing both the error rate and the number of errors with no performance overhead. In this spirit, we propose a metric Reliability Performance Ratio to measure the performance-reliability tradeoff achieved by a reliability mechanism.

Weaver et al. [2] proposed Mean Instructions to Failure (MITF) to quantify the tradeoff between performance and reliability. MITF is calculated as:

$$\begin{aligned} \text{MITF} &= \left(\frac{number\ of\ committed\ instructions}{number\ of\ errors}\right) \\ &\propto \left(\frac{Instructions}{ACE\ Cycles}\right) \end{aligned}$$

If the number of instructions is constant, then an increase in MITF only implies a reduction in ACE cycles, and it does not capture the cost incurred in execution cycles to achieve this reduction. We propose a new metric, Reliability Performance Ratio (RPR), to capture the impact of a design parameter on both performance and reliability, and is analogous to Energy Performance Ratio (EPR) proposed by Hofstee et al. to study the

tradeoff between power consumption and performance [151]. In this paper, we measure RPR of a design parameter as the ratio of the percentage decrease in ACE cycles to the percentage increase in execution cycles, as shown by the equation:

$$\text{RPR} \quad = \quad \left(\frac{Percentage\ decrease\ in\ ACE\ cycles}{Percentage\ increase\ in\ execution\ cycles} \right)$$

Thus an RPR of 1 for a parameter means that the ACE cycles improves by 1% for a cost of 1% in execution time. An RPR of 2 is better than an RPR of 1 since it implies a 2% increase in reliability for the same 1% decrease in performance. If two design parameters have different RPRs, designers can exploit this asymmetry to improve reliability at constant performance by reducing the design parameter at lower RPR, while maintaining the performance by increasing the parameter with higher RPR. Thus RPR provides the foundation for a systematic approach where designers work to achieve the reliability target, at the same time optimizing the RPR to achieve low overhead.

In practice, there are some important caveats to consider when using the RPR metric. First, the RPR of a single parameter may vary depending upon the trend in execution time and ACE cycles, as more of the design parameter is used. For instance, the RPR of pipelining can be quite different at the beginning when it is very beneficial, from the point at which pipelining starts to provide diminishing returns in performance. So the RPR of a technique may can be different in different regions of this curve. In fact, RPR can become infinity on the graph if at any point there is a change in ACE cycles, with no associated change in the execution time. However, these discontinuities in the graph will be rare, and do not correspond to the average RPR of the parameter in that region. Second, quantization limits of the technique that only allow discrete points on this curve, and design constraints besides reliability may force us to consider the absolute impact on the different parameters such as performance, reliability, area, and power consumption and give less

importance to the RPR itself. Finally, it is possible that a technique improves both system reliability and performance. For instance, prefetching state into ECC or parity protected memory structures can reduce the execution time of the program without adding any vulnerable state. Further, the smaller execution time can also lead to a reduction in total ACE cycles. These techniques are most desirable from an RPR viewpoint because they improve reliability without adding any performance overhead.

6.4 Soft Error Reliability Regimes

This section presents the four architectural soft error reliability regimes. Three of the regimes, error correcting codes, full redundant execution (FRE), and selective redundant execution (SRE) are based on fault tolerance; and the last regime, AVF throttling, is based on fault evasion.

6.4.1 Error Correcting Codes

The first regime is based on incrementally adding parity or ECC protection to on-chip structures to keep the SER/chip within the reliability budget. With ECC, for 64 bit data 7 additional bits are needed to correct all single-bit errors [2]. While this provides near-zero SER for that structure, it adds an overhead of 11% in the number of bits. Further, ECC also requires extra circuitry on both the read and the write datapath for checking and generating the error code. Tyler et al. [21] calculated the delay through an optimized ECC generation circuit to be between 8-10 fan-out-of-4 inverter (FO4) gate delays. This takes up almost 50-80% of the 12-16 FO4 cycle time target in a high performance microprocessor, and thus will either potentially extend the cycle time or add extra cycles. While the overall area and timing overhead is amortized for large structures such as caches, in-line ECC checking or generation may be unacceptable in on-chip storage structures such as instruction windows that are tightly integrated into the pipeline, and for which instruction read or write is performed serially with several other operations such as decode, wakeup or

selection in the same pipeline stage. While ECC generation or checking can also be performed in a pipelined fashion, this adds datapath complexity to eventually trace and flush the corresponding data before it can update architectural state [2].

Parity has smaller area overhead due to the extra storage and circuitry, since it only provides detection capability. However, errors which might have earlier been masked and not affected program output, are now reported as parity failures leading to an overall increase in error rate. Section 6.2 listed several factors that can lead to false positives, including parity errors in processor state that is ex-ace, mis-speculated, logically masked, predicated-false, or dynamically dead. Mukherjee et al. showed that false parity errors account for more than 52% of the total errors, and naively adding parity to the instruction window more than doubles the error rate by worsening the AVF from 29% to 62% [2]. Based on this result, they proposed tracking parity errors until they can be determined to be a true or a false error, at which time only the true errors are allowed to interrupt the system while the false errors are filtered. They described the logic required for correctly propagating parity errors through different execution and memory operations for detecting false positives. For instance, they observe that while a parity error on mis-speculated state can be filtered at instruction retirement with modest extra hardware, detecting false errors due to dynamically dead state will require storing and analyzing instructions in a post-retirement buffer with at least 512 entries to achieve significant coverage.

6.4.2 Full Redundant Execution (FRE)

While error detecting and correcting codes can be applied to storage structures, and can be extended to regular datapath logic [152], control logic is typically unprotected by these mechanisms. Since FRE in general, redundantly performs all the operations in all the pipeline stages for each instruction, it achieves protection over both the datapath and control logic. Hence, while FRE provides near-zero (SER/chip), it may incur a significant cost in performance, power, area, and complexity. There have been several different approaches

implemented and proposed in research for improving the coverage and reducing the overhead of FRE [17, 19, 25, 92, 153].

Systems with a long history of implementing high reliability through redundant execution include IBM S/360 that is currently the zSeries [18,154], the HP NonStop Server [153], and the Stratus continuous processing technology [25]. The scope of a faulty operation is quickly contained, either by using duplicated processor pairs run in lock-stepped fashion as in the case of the HP NonStop Server and Stratus, or by using duplicate pipeline stages for fault detection, and a checkpoint and instruction retry for transparent recovery, as in the case of IBM zSeries systems. A comprehensive overview and comparison of the IBM and HP fault tolerance approaches is provided in [153].

There have been several approaches proposed in research for further improving the efficiency of redundant execution. DIVA achieves extremely low performance overhead by using a simple dedicated checker to verify the results of instructions ready to be committed by the high performance core [17]. The checker is a standard five-stage in-order processor designed with sufficiently large transistors and operated at a clock rate sufficient to make it immune to soft errors. Despite its slow clock rate and simple design, the checker does not become a bottleneck because it does not incur mis-speculation penalties and incurs virtually no memory system overhead due to the prefetching effect caused by the high performance core. Since the recomputations have both a spatial and temporal gap they will not be affected by the temporal or spatial locality of the particles. Researchers have suggested modifications to the superscalar hardware to support redundant execution for a subset of the pipeline stages [155, 156]. There have also been proposals to leverage the multi-threading support in hardware or software to implement redundant execution [19, 90–92]. Both AR-SMT [90] and SRT [91] use a hardware mechanism called simultaneous multithreading to drive the redundant threads of execution. Both these schemes are rather complex, but SRT has the advantage that it does not require changes to the operating system and can handle multi-cycle faults. Recent research has explored the possibility of exploiting

the redundancy in CMPs for light-weight redundant execution [127].

Redundant execution has also been applied at the circuit or logic level through concurrent error detection techniques [157]. Recently, Mitra et al. proposed built-in soft error resilience (BISER), a temporal redundant execution technique that uses the assistance of already existing design for test (DFT) structures [158]. This mechanism elegantly reuses already existing scan latches for fault detection, uses a C-element for fault recovery, and achieves high reliability with minimal overhead in area and power consumption. Since the scan chains in a processor are generally used at a slow frequency, they are typically sized smaller than normal latches to minimize the overhead in area and power consumption. However, BISER requires the scan latches to be operated at the processor clock frequency for fault detection. The challenge here lies in enabling at-speed use of the scan chains, and at the same time balancing the fault coverage with the overheads in area and power consumption.

6.4.3 AVF Throttling

FRE proactively detects faults in the execution by redundantly performing all instruction operations at the cost of higher complexity, power consumption, and performance overhead. In this dissertation, we propose a new reliability regime, AVF throttling, based on fault evasion rather than fault tolerance. AVF throttling is at the other end of the spectrum in comparison to FRE and completely eliminates redundant execution. It improves reliability by trading concurrency to reduce the amount of processor state vulnerable to soft errors.

Modern microprocessors employ several concurrency techniques to improve performance. Many of these techniques such as speculative execution, achieve higher performance by aggressively bringing future program state into the processor and mining them for available parallelism. As these techniques increase the average time for which program state is resident on the processor, they also increase the probability of the state being corrupted by a soft error and hence the structure's AVF. The performance-reliability tradeoff

therefore refers to this opposing interaction between the larger AVF and the smaller execution time. A key observation is that while exploiting parallelism along the longest program path can significantly reduce execution time, fetching program state into the processor corresponding to the majority of other program paths can be delayed by several cycles without any effect on the execution time. Fields et al. termed this latency tolerance exhibited by program state as *slack*, and demonstrated that there is a significant amount of slack in program execution [159]. The fault evasive mechanisms we explore later in this dissertation precisely take advantage of this abundant execution slack to defer the fetch of program state, thus reducing the amount of vulnerable processor state, and at the same time also minimizing the performance overhead.

AVF throttling mechanisms thus reduce processor ACE cycles by reducing both the vulnerable processor state and exposure time to soft errors. AVF throttling is built upon two important observations.

Protected vs unprotected storage: While structures such as instruction and data caches are typically protected, other processor structures such as the reorder buffer, register file and latches may not always be protected. So when state is aggressively fetched out of protected structures, it is effectively converted from protected to vulnerable state, and thus now begins accumulating ACE cycles.

Error expansion: It is also common that state stored in microarchitectural structures are wider since they include either post-decoded information or additional pipeline control state. For instance, while instructions in the cache are typically 32 bits wide, a post-decoded instruction in the instruction window may well be 100 bits wide. Therefore, the same information now accrues more ACE cycles and has a higher chance of being corrupted.

The work that comes closest to AVF throttling is the concept of triggers and actions proposed by Weaver et al. [2]. They observe that in an in-order processor, events such as L2 misses increase the vulnerability of the instructions waiting in the instruction window that

are behind this L2 miss. They proposed to reduce the ACE cycles accumulated by flushing the instruction window on L2 misses. They showed that this approach incurred minimal performance overhead in an in-order architecture. Qureshi et al. [160] proposed to use the slack available on an L2 miss to perform redundant execution of instructions to reduce SER. While both these techniques manage the originally available slack, the AVF throttling regime we propose aims to reduce slack by spreading the computation appropriately and fetching program state just when needed. While deferring the fetch of program instructions has been proposed earlier by Muthler et al. for improving performance [161], we use it to improve soft error reliability. Reducing instruction window SER using front end throttling was also explored by Kalappurakkal et al. [162].

6.4.4 Selective Redundant Execution (SRE)

Selective redundant execution can be viewed in primarily two different ways. First is the hardware centric view, where redundant execution is performed only in a subset of the pipeline stages. IBM System 390 used redundant execution based fault tolerance only in the decode and execute stages of the processor [18]. There is some area overhead due to duplication, and extra power consumption due to redundant execution in these stages, but low performance overhead.

Second is the program centric view, where redundant execution is performed only for selective portions of the program [118, 119]. Parashar et al. exploit high confidence control flow, address, and value predictions as a means for verifying the corresponding branch, load, and store addresses and values, and do not redundantly execute the program slices computing these values [118]. Gomaa et al. dynamically detect periods of low single-thread performance during which they trigger redundant execution to reduce performance overhead [119]. These and other hardware proposals for doing SRE arguably achieve lower fault coverage than full redundant execution (FRE), but have the potential to significantly improve the performance overhead. However, it also increases the hardware complexity, be-

cause hardware now not only must perform redundant execution, but must also dynamically decide when to and when not to trigger redundant execution.

Other research has explored software-based SRE, and investigated using offline profiling and optimization algorithms to determine the degree of protection required in different parts of the program [117]. In this approach, the compiler uses this information to appropriately insert redundant instructions in the program. For instance, if the analysis revealed that only the control flow requires verification for a particular section of the program, only the slice of instructions that determine the control flow are duplicated. While software takes almost the full responsibility for redundant execution in this methodology, an alternative hybrid methodology may use the compiler only to annotate different parts of the program using different execution flags, and not actually insert the redundant instructions. The execution flags may denote different execution modes including but not limited to normal execution, redundant execution, and redundant execution of only the memory instructions, and the hardware can then interpret these flags dynamically to trigger the appropriate form of redundant execution.

Partial redundancy is also applicable at other levels of the design. Cost-effective partial duplication of logic gates is motivated by an observation of electrical masking, and duplicates only the gates nearer the latch that are more susceptible [10]. Selective capacitance insertion focuses on increasing latching-window masking to reduce the probability of latching transient errors [11]. While device or circuit level techniques directly target the actual source of a fault, higher level mechanisms using redundant execution and the compiler may be able to provide better reliability guarantees and lower overhead at the system level.

6.4.5 Summary

While FRE can be implemented using redundant instructions within a single thread, as two threads on the same processor, or as two threads on two different processors, it incurs high overhead in all cases [19]. Hence, it will be justified only for worst-case soft error

rate scaling trends or for markets with very stringent reliability requirements, and will be over-designed in other cases. On the other hand, AVF throttling is a class of microarchitectural fault evasion techniques that efficiently reduces the amount of vulnerable processor state. SRE uses hardware or software approaches to redundantly execute only specific subportions of the program. Therefore, both AVF throttling and SRE can significantly improve the performance-reliability tradeoff over FRE. Further, AVF throttling is significantly simpler than both FRE and SRE since it is not based on redundant execution, and hence is also more power efficient.

While FRE or SRE based fault detection will probably be required to achieve the AVF reduction in the first two scaling scenarios in Figure 6.2, and ECC can be applied to large storage structures, we propose that microarchitectural fault evasion techniques such as AVF throttling, can achieve the 5%-25% AVF reduction required in last two scaling scenarios at lower overhead. Further, similar to SRE, the performance-reliability tradeoff of AVF throttling can also be configured based on the application and reliability requirements. Chapter 7 extends this analysis, and presents a detailed comparison of the reliability improvement, performance, area, power consumption, and complexity overheads of the four regimes.

6.5 TRIPS AVF Methodology and Evaluation

This section presents the detailed methodology for evaluating the AVF of important microarchitectural structures in the TRIPS processor using ACE analysis. Chapter 4 described the TRIPS microarchitecture in detail, and Table 6.1 lists the TRIPS processor structures for which we computed the ACE cycles, along with the number of instances in the TRIPS core, and the number of entries in each instance. Features such as extensive predication, fewer architectural register file reads/writes, and the physical partitioning of all major structures have the potential to reduce the per-bit AVF in comparison with conventional architectures. At the same time, the block-atomic execution model which has the potential to extend the

Unit	Register Tile			Execution Tile		Data Tile		Operand Router
Structure	Register file	Read queue	Write queue	Instruction buffer	Operand buffer	LSQ buffer	LSQ CAM	OPN buffer
Instances	4	4	4	16	2x16 (opA, opB)	4	4	100
Entries	32	64	64	64	64	256	256	4

Table 6.1: TRIPS processor structures for which ACE cycles are tracked

occupancy of block state on the processor, and the larger sizes of processor structures including but not limited to the instruction window and operand buffers have the potential to increase the overall vulnerability of the structure over conventional architectures. The processor ACE cycles presented in the rest of the dissertation is the sum of the ACE cycles across all these structures. In computing the ACE cycles and the AVF, we account for the sources of un-ACE cycles that are applicable from the list in Section 6.2.

6.5.1 TRIPS AVF Methodology

The following paragraphs give a detailed explanation of the different un-ACE factors that were considered for each structure in Table 6.1 in computing the AVF.

Instruction Buffer in the Execution Tile: Figure 6.3 shows the TRIPS instruction formats. Since the *read* and *write* instructions are stored in the RT, they are not considered here. While Figure 6.3 shows the format of 32 bit pre-decoded instructions, each entry in the instruction buffer contains a post-decoded instruction which is 74 bits wide. In addition to the opcode bits, predication mode specifier bits, branch offsets, data for immediate and constant operands, and destination target specifiers, a post-decoded instruction also contains control bits that encode the instruction type, that aid in functional unit and operand selection during execution, and that speed-up the construction of the OPN control and data packet that are sent to the router.

Since the TRIPS compiler does not currently generate any performance enhancing instructions, our analysis does not include masking due to that factor. We also restricted our

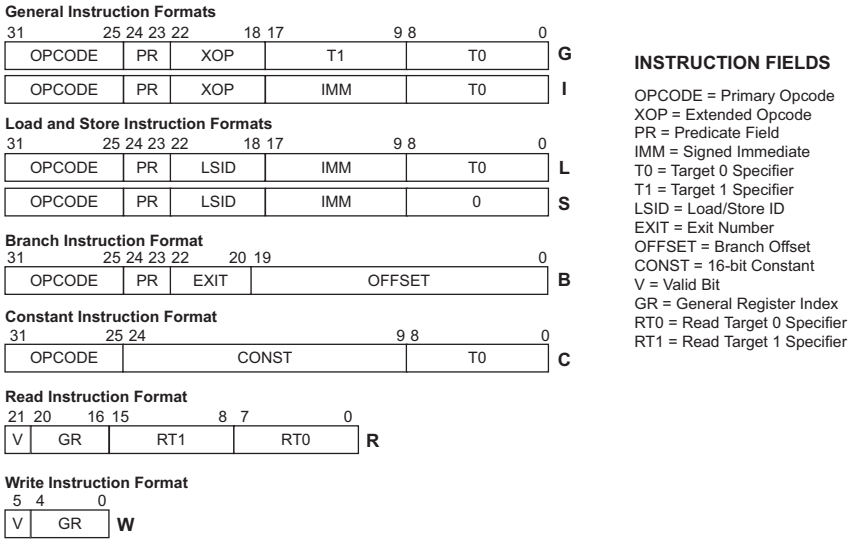

Figure 6.3: TRIPS instruction formats

analysis of logical masking to bitwise logical operations, and sign extension instructions which are encoded in the general instruction format. Masking due to NOP, invalid, and ex-ace state are easily computed by tracking valid block instructions, block commit and flush events, and the time when each instruction enters into, and is issued out of the instruction buffer for execution, respectively. The destination target specifiers, and the control bits related to OPN control packet formation are not derated for dynamically dead instructions. The predicate specifier bits are not derated for the predicated-false instructions. Additionally, we do not derate the target specifier bits of indirectly predicated-false instructions, instructions which are themselves not predicated but which deliver data to those that are predicated-false.

Operand Buffer in the Execution Tile: The data values in the operand buffer are 64 bits wide. In computing the AVF, we account for invalid state, mis-speculated state, ex-ace state, predicated-false instructions, dynamically dead instructions, and logical masking. Since the information contained is only data, the vulnerable cycles are fully derated for all un-ACE state, including predicated-false and dynamically dead instructions. We model relatively

Valid	Register Specifier	Target 1	Target 2	Write Queue Entry Valid	Write Queue Entry Index	Write Queue Entry Issued	Read Queue Entry Issued
1 bit	7 bits	9 bits	9 bits	1 bit	5 bits	1 bit	1 bit

Table 6.2: Bit fields of a read queue entry

simple support for logical masking and only account for bitwise logical instructions, and sign extension instructions.

Architectural Register File in the Register Tile: The architectural register file contains committed program register state, and each register is 64 bits wide. Since it contains committed state, only a subset of the masking factors are applicable for the register file. For instance, it cannot contain state that is invalid, mis-speculated, or produced by predicated-false instructions. On the other hand, we account for the register value being in ex-ace state after it has been read for the last time by a block that is not on the mis-speculated path, and before it is over-written with a new value. We also derate register data that is over written before it is read, since it is dynamically dead. Finally, we account for some of the bits in the register value being logically masked, if they are logically masked in all of their uses.

Read Queues in the Register Tile: Read queues and write queues are used for register forwarding in the TRIPS processor. The write queue entries buffer the block register outputs until they are determined to be non-speculative when they can be committed to the architectural register file. Each read queue entry contains the register specifier denoting the architectural register to read, using which it matches against the writes to the same register buffered in the write queue from older blocks. Register forwarding from the write to the read queue uses several control bits in the read queue including the issued status, the pointer to the write queue entry, and the status of the write queue entry, to signal the current status of forwarding. The read queue entry also contains two target specifiers pointing to the instructions to which the register value must be sent. Table 6.2 shows the bit fields of a read queue entry.

Valid	Register Specifier	Data	Ready	Value Forwarded to Read Queue	Exception	Null	Committed
1 bit	7 bits	64 bits	1 bit	1 bit	1 bit	1 bit	1 bit

Table 6.3: Bit fields of a write queue entry

The valid bit of an entry is always considered to be ACE since it can either lead to a register value not being delivered to the block if a valid bit becomes invalid, or lead to incorrectly delivering a register value to the block if an invalid entry is marked valid. The susceptibility of the register and target specifiers, and the read queue entry issued bit are dependent on whether the entry is valid, since they are vulnerable only during the window when the entry is valid. Similarly, other control bits such as the issued status of the write queue entry, and the write queue entry index from where the value is delivered, are vulnerable only when the control bit indicating whether there is a valid write queue entry corresponding to this read queue entry is set. Once the vulnerable windows for the different bit fields are computed, they are then used in a conventional fashion to apply further derating using mis-speculated information, ex-ace information, and dynamically dead information.

Write Queues in the Register Tile: The write queues store the actual data to be committed to the architectural register file. Table 6.3 shows the different fields in a write queue entry. While information in the read queue entry is generally vulnerable only until it is used (after which it is ex-ace), the data in the write queue entry is vulnerable until it is successfully written into the register file. This is the primary difference between read queue and the write queue entry vulnerability. Due to this difference, all the fields of the write queue are vulnerable for the entire duration when the entry is valid. Using a similar methodology, the vulnerable window is used to apply further derating due to ex-ace state, speculative state, and dynamically dead state to estimate the AVF.

LSQ structures in the Data Tile: The load-store queues are responsible for maintaining program order and enabling data forwarding between dependent loads and stores to improve

Name	Description
CTRL_VALID	Indicates whether control packet contains valid information.
CTRL_TYPE[3:0]	Message type: Identifies the type of message 0 - Generic Transfer or Reply 1 - Load Request 2 - Store Request 4 - PC Read Request 5 - Branch/PC Write
CTRL_BID[2:0]	Block ID: Internal hardware or program block.
CTRL_DNODE[5:0]	Destination tile: Identified as a 6-bit (xxxyyy) coordinate.
CTRL_DINDEX[4:0]	Destination Index: used by the destination tile.
CTRL_SNODE[5:0]	Source node: Identified as a 6-bit (xxxyyy) coordinate.
CTRL_SINDEX[4:0]	Source Index: used for debugging.

Table 6.4: Bit fields of an OPN control packet

performance. The AVF for the loads and stores are calculated independently since they exhibit different behavior. The vulnerable window for the load address is from the time the address is available in the LSQ, to the last time it is used to gather the data. The load data in the LSQ can be obtained from prior stores in the LSQ, and/or the data cache, and/or the miss-handling logic (MSHRs), and needs to be stored in the LSQ only until it has been fully constructed and can be sent to the target instruction. On the other hand, the address and data of store instructions are vulnerable until they are successfully committed to the caches, similar to register data in the write queues. We further take into account derating due to load and store instructions that are mis-speculated, directly or indirectly predicated-false, and dynamically dead.

Operand Buffers in the Operand Router: The operand router is responsible for routing data between the different tiles in the TRIPS processor. It uses a protocol where each data packet is preceded by a control packet, and the control packet is always exactly one cycle ahead of the corresponding data packet [110]. Every tile in the TRIPS processor has a router, and each router has separate control packet and data packet buffers for each of the four directions (North, South, East, West). We compute the AVF of the control and data

Name	Description
DATA_VALID	Indicates whether data packet has valid information. Cancelled or flushed transfers are invalidated by the sender.
DATA_TYPE[1:0]	Identifies the type of data being delivered 0 - Normal 1 - Null 2 - Exception
DATA_VALUE[63:0]	Message payload Unused bits should be assigned to logic 0. DATA_VALUE[2:0] exit number for branches, load target for loads DATA_VALUE[8:0] load target for loads, PC read requests
DATA_ADDR[39:0]	In the normal operating mode, this field contains the 40-bit virtual address for memory operations or the PC value for pc writes
DATA_OP[2:0]	Lower bits of opcode for branches, loads, stores; for other cases, this field is unused (set to 0's).

Table 6.5: Bit fields of an OPN data packet

packet buffers in all the tiles.

Table 6.4 shows the format of the OPN control packet. Control packets of multiple types are supported indicative of the instruction type that generates them. The Block ID identifies the TRIPS block to which these packets belong. Finally, the source node and index, and destination node and index identify the source and destination locations for this packet. Since the packet leaves the buffer immediately after it is processed, there is never any ex-ace state associated with the buffer. If the control packet belongs to a block that is mis-speculated, all bit fields of the control packet are derated except the Block ID which is used for flushing the packet. Our methodology currently does not account for derating control packets corresponding to predicated-false instructions and dynamically dead instructions.

Table 6.5 shows the format of the OPN data packet. The TYPE field indicates whether the data packet is normal, null, or has an exception. As shown in the table, the size of the data value, address, and op fields depend on the actual type of instruction generating the data packet. For instance, while a load or store instruction uses both the data address and value fields, a general format instruction such as ADD uses only the value field. Therefore,

based on the instruction type we determine the actual number of meaningful bits in the data address, value, and op fields and only take those into account in computing the AVF. Similar to the control packet, there is never any ex-ace state in the data buffers. All the fields are derated if the data packet is mis-speculated. Our methodology does not account for derating data packets corresponding to predicated-false instructions and dynamically dead instructions.

6.5.2 TRIPS Baseline AVF Results

We augment the cycle-accurate, validated execution driven simulator that models the TRIPS architecture in detail to compute the ACE cycles. We used 24 compiler-generated and 7 hand-optimized benchmarks from the EEMBC 2.0 suite, 10 benchmarks from the SPEC 2000 floating point suite, and 8 benchmarks from the SPEC 2000 integer suite. All the SPEC benchmarks are generated by the compiler. The rest of the methodology is identical to that described in Section 5.3.4. More details regarding each of these structures can be found in [108].

Table 6.6 presents the AVF for the different structures analyzed. The first four columns present the AVF for the individual benchmark categories. At a high level, the hand-optimized benchmarks have the highest AVF, followed by the compiler-generated SPEC floating point benchmarks, EEMBC benchmarks, and finally the SPEC integer benchmarks. This is not surprising since the order corresponds to the expected degree of processor utilization. However, in contrast to results form prior research [123], the average AVF of the instruction window for SPEC floating point benchmarks is less than for SPEC integer benchmarks. This is primarily because floating point instructions stay for a shorter time in the instruction window in comparison to integer instructions due to the higher inherent parallelism available in floating point programs.

The last three columns of Table 6.6 present the AVF averaged across all the benchmark categories at two granularities; AVF, which represents the error probability per bit of

Structure	Hand optimized AVF	EEMBC AVF	SPEC INT AVF	SPEC FP AVF	Average AVF	Average (AVFxBITS)	Average Percentage Contribution
Register File	11.9%	10.5%	7.8%	12.1%	10.6%	866.2	7.4%
Read Queue	5.7%	4.6%	4.1%	5.4%	4.8%	419.2	3.6%
Write Queue	5.1%	3.8%	3.2%	7.2%	4.6%	918.1	7.8%
Instruction Buffer	15.5%	10.1%	9.4%	6.2%	9.8%	7453.1	63.5%
Operand Buffer	1.6%	0.7%	0.5%	0.8%	0.8%	1037.8	8.9%
LSQ Buffer	0.2%	0.4%	0.3%	0.6%	0.4%	255.0	2.2%
LSQ CAM	1.1%	0.9%	0.5%	1.5%	1.0%	411.8	3.5%
OPN Buffer	0.9%	0.5%	0.4%	1.0%	0.7%	368.2	3.1%

Table 6.6: Baseline AVF results

the structure per cycle, and (AVF x BITS), that indicates the error probability of the overall structure across all the tiles in which the structure exists, per cycle. For instance, (AVF x BITS) of the local instruction buffer accounts for the bits in all the 16 distributed instruction buffers in the processor. The same data is also shown normalized to the total (AVF x BITS) to show the relative contributions of the different structures to the overall processor SER.

The architectural register file, and the local instruction buffers at each ET have the maximum AVF. These results are consistent with prior research that have reported the highest vulnerability for these two processor structures [92, 123]. However, the absolute values of the AVF for these two structures are considerably smaller in TRIPS in comparison to prior research that reported an AVF of 18.65% for the register file [92], and 28% for the instruction window [123]. The primary reason the register file has lower AVF is because the TRIPS processor implements a storage hierarchy for managing instruction operands to optimize for capacity, latency, and bandwidth. The hierarchy consists of three levels: the architectural register file, and the read and write queues that perform register forwarding between blocks in the RTs, and the local operand buffers at each ET. As mentioned before, accesses within a block only need to access the local operand buffers, while the read, and write queues and the register file are accessed at block boundaries. This division of accesses, along with the extensive partitioning of all the structures that spreads the accesses

physically, together reduce their vulnerability when compared to conventional architectures. Similarly, the extensive partitioning of the instruction window reduces its average vulnerability. Further, the large capacity of the window allows more speculative instructions which are inherently less vulnerable.

Accounting for the structure capacities using (AVF x BITS) shows that many structures including the Write Queues used for register value forwarding, and the local operand buffers at each ET have vulnerabilities greater than the register file. Reis et al. reported an AVF of 18.65% for a register file with a total capacity of 8320 bits (128 registers x 65 bits) resulting in an (AVF x BITS) of 1539.5. Although the capacity of the TRIPS register file is similar (128 registers x 64 bits), the (AVF x BITS) is smaller than this because of the distribution of accesses across the three levels of the hierarchy. Computing the contribution of all the three levels of the hierarchy by adding the (AVF x BITS) of the register file, read queue, write queue, and the local operand buffers produces a total of 3241.3. While the combined capacity of these structures is almost 20 times greater than the register file in the conventional architecture, the overall hierarchy is only twice as vulnerable. The vulnerability does not scale with capacity because the logical partitioning of the accesses, and the physical partitioning of the structures reduce the occupancy and utilization. Further, as mentioned, the vulnerability is also a function of all the other masking factors enumerated in Section 6.2. Similarly, Mukherjee et al. reported an AVF of 28% for an instruction window with a total capacity of 6400 bits (64 entries x 100 bits) resulting in an (AVF x BITS) of 1792. While the TRIPS distributed instruction window with a capacity of 75776 bits (1024 entries x 74 bits) is more than 10X larger, it has an (AVF x BITS) of 7453.1 which makes it about four times more vulnerable than the conventional instruction window. The vulnerability again does not scale with capacity because it depends on all the factors enumerated in Section 6.2.

Together the register file, read and write queues, and the local operand buffers at each ET contribute 27.7% of the overall ACE cycles. The instruction window is still the dominant contributor accounting for 63% of the processor error rate. This is partially be-

cause instructions have more control information that cannot be derated easily when compared to data. Further, instructions in the TRIPS processor arrive much earlier than the operands due to the block-atomic execution model, and the large instruction window and hence accumulate more ACE cycles. Chapter 7 provides more insight into this analysis. In summary, while physical and logical partitioning of the structures and accesses reduces the per-bit AVF, the larger capacity and the greater amount of state increases the overall vulnerability of the structures. However, the 10-20X increase in capacity only leads to a 2-4X increase in vulnerability. Prior research has demonstrated that these larger capacities can be used used to out-perform conventional architectures by a factor of 3X on a highly optimized set of benchmarks [163], providing an RPR of approximately 1. In the future, such analysis of the performance-reliability tradeoff will be an important component of the design process along with studying the power consumption, area, and complexity tradeoffs.

6.6 Discussion

The most widely used methodology for estimating AVF is called statistical fault injection (SFI). Statistical fault injection works by randomly injecting bit flips into the bits of the processor structures being studied, and then comparing the architectural state of the model with a golden reference model after simulating for a certain number of cycles. While a mismatch indicates an error, a successful match can either mean that the error has been masked or is still latent in the processor. A more detailed comparison of the microarchitectural state can reduce the probability of the error still being latent and improves the accuracy of the analysis [122]. Multiple fault injections are performed for each structure to attain statistical significance based on an analysis of confidence intervals. Both the time interval between two fault injections and the physical location of the fault injection are chosen randomly for every experiment. The fraction of runs for which the outputs mismatch represents the AVF for the structure. While SFI can be used both on a performance model and at the RTL level, its accuracy is higher at the RTL level where the microarchitecture is modeled in greater

detail.

Mukherjee et al. analyzed the advantages and disadvantages of the two techniques, ACE analysis, and SFI in computing AVF [123]. While SFI does not require a detailed knowledge of the underlying architecture, the length of each fault injection run is usually limited to tens of thousands of cycles because RTL simulation is very time consuming. Further, estimating the AVF of a structure requires multiple fault injection experiments to achieve statistical significance and is very computationally intensive. On the other hand, ACE analysis can measure the AVF for all the processor structures simulated by the performance model using just one detailed simulation. Wang et al. argue that due to several reasons including the abstract nature of the performance model, SFI will provide significantly greater accuracy than ACE analysis [164]. However, since the performance models used in industry model the microarchitecture in great detail, ACE analysis should provide reasonably good accuracy, and in much less time than SFI. In summary, since performance models are available earlier in the design than RTL, ACE analysis allows a designer to estimate processor reliability and make important architectural decisions for improving reliability, earlier in the design cycle, albeit with lower accuracy. Further, ACE analysis is ideal for this process since it permits a better understanding of the interaction between the architectural decisions for high performance and their impact on reliability. For instance, ACE analysis can be used to determine the relative contributions of different processor structures to the overall ACE cycles, or to study the effect of architectural parameters such as branch misprediction rate or the percentage of dynamically dead instructions on the AVF.

6.7 Summary

In this chapter, we extend the raw soft error rate analysis performed in Chapter 3 to the chip level by taking into account architectural masking factors. We use the methodology proposed by Mukherjee et al. to compute the architectural vulnerability factors for critical TRIPS processor structures. We extend the methodology to build a quantitative framework

for analyzing the performance-reliability tradeoff. We propose Reliability Performance Ratio (RPR), a new metric which a designer can use to measure the performance-reliability tradeoff, and achieve the reliability target with the least excessive protection and overhead. We analyze the scaling trend of chip soft error rate to quantify the soft error reliability improvement needed every technology generation, and discuss the performance-reliability tradeoffs of achieving it at different levels of the design including device techniques, error correcting codes, and microarchitectural mechanisms. Based on this analysis, we propose a classification of the architectural soft error reliability techniques into four regimes each with a different performance-reliability tradeoff. In the next chapter, we explore mechanisms from each of the regimes and present a detailed comparison of the reliability improvement and the associated tradeoffs.

Chapter 7

Techniques for Improving Soft Error Reliability

This chapter presents techniques from three soft error reliability regimes: full redundant execution (FRE), AVF throttling, and selective redundant execution (SRE). As described in Table 4.4, the techniques are built upon key features of the TRIPS architecture: tile-based modular design, on-chip networks, block-atomic execution model, and static instruction placement to lower overhead. Sections 7.1, 7.2, and 7.3 describe the techniques and present the results for FRE, AVF throttling, and SRE respectively. For each regime we evaluate the reliability improvement as the percentage decrease in ACE cycles, and the performance overhead as the percentage increase in execution cycles relative to the baseline TRIPS microarchitecture in normal execution mode using all the execution resources. Finally, Section 7.4 provides a detailed comparison of the reliability improvement, performance overhead, and other design tradeoffs associated with the regimes.

As described in Section 5.3.4, the execution cycles and ACE cycles are both computed using a cycle-accurate, validated execution driven simulator that models the TRIPS architecture and the distributed protocols in detail [111]. We use 24 benchmarks from the EEMBC 2.0 suite, 10 benchmarks from the SPEC 2000 floating point suite, and 8 bench-

marks from the SPEC 2000 integer suite. Apart from these compiler-generated benchmarks, we also use 7 hand-optimized EEMBC benchmarks to account for the fact that the TRIPS compiler is still in development. As shown in Table 5.7, the hand-optimized benchmarks have significantly larger block sizes and hence utilize the processor resources more effectively.

7.1 Full Redundant Execution (FRE)

The TRIPS block-atomic execution model provides two basic forms of redundant execution: redundant execution of instructions within a block, and redundant execution of blocks at the block granularity. Full redundant execution within a block basically requires each block instruction and all its operations in the pipeline (fetch, decode, register read, execute, writeback) to be duplicated within the block. The block size has to be appropriately reduced to allow space for this duplication. In the latter model, each TRIPS block is associated with a *corresponding redundant block* (CRB) that verifies its execution. In this dissertation, we explore this latter model of redundant execution at the block granularity because it has some qualitative advantages. First, block-granular redundant execution is a natural fit for the block-atomic execution model and is independent of block utilization. Second, the TRIPS microarchitecture easily lends itself to this model since the majority of the processor queues are already statically block-partitioned (see Section 4.2.1). Third, using a primary and a redundant block provides greater opportunity for slack between the redundant operations for higher reliability. Finally, fault detection at the block granularity amortizes the overhead of comparing the outputs of each instruction and allows better scalability at future technologies. While we primarily focus on fault detection, TRIPS blocks provide a natural boundary for checkpointing, and the model can be naturally extended to include fault recovery. Figures 7.1 and 7.2 illustrate our FRE model. The following paragraphs describe the model and the additions needed to the baseline TRIPS microarchitecture (explained in Section 4.2) in greater detail.

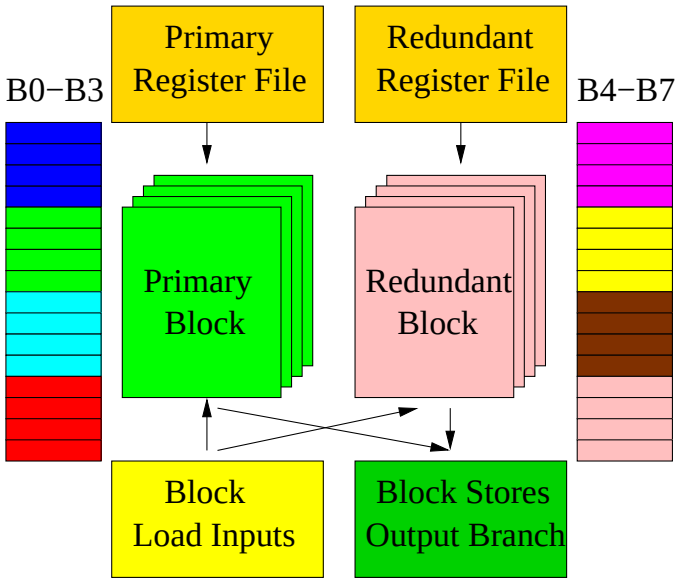

Figure 7.1: Block-granular redundant execution model

Redundant Execution Model: In the baseline microarchitecture, the 64 local instruction and operand buffer entries at each ET are statically divided into eight block partitions allowing simultaneous execution of up to eight blocks. For redundant execution, these entries are statically and equally divided to allow up to four primary and redundant blocks to execute simultaneously. Resources that are not statically allocated such as the functional units are dynamically shared between the primary and redundant blocks. If events from both the primary and the redundant block are ready in the same cycle, the model uses a simple round-robin scheme for arbitration. For instance, primary block and CRB commits use round-robin scheduling. The baseline microarchitecture already contains round-robin thread scheduling logic for use in multi-threaded mode, which can be augmented for use in redundant execution mode.

Multiple CRBs execute simultaneously, similar to the primary blocks, to reduce performance overhead. Further, only primary blocks that complete successfully without any mispredictions are redundantly executed, and mis-speculated primary blocks are ignored.

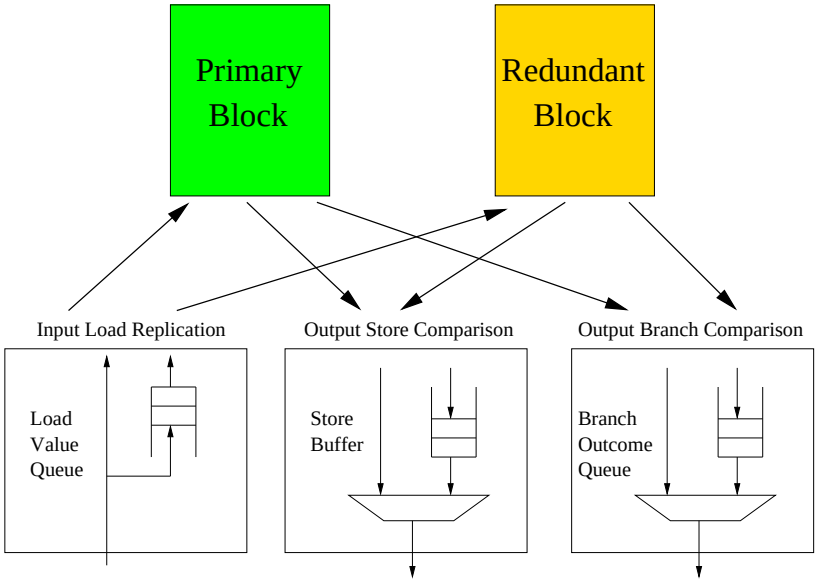

Figure 7.2: Using the LSQ to perform replication of load values and comparison of stores, and the BOQ to compare branch addresses

This reduces the number of instructions that are redundantly executed and decreases contention for the shared execution resources. While CRB instructions can begin execution only after successful completion of the primary block, CRB allocation, instruction fetch from the I-cache, and delivery of operands from older CRB's, are initiated as early as possible to effectively hide their transmission latency. CRB instructions and operands thus fetched occupy their allocated instruction and operand buffer entries, and wait for primary block completion. The commit commands on the GCN, which inform every instruction of the oldest primary block (at each of the ETs) about its successful completion, are augmented to trigger the execution of the CRB. A CRB instruction that is otherwise ready for execution can be selected only after the notification of primary block completion is received.

Detecting Control Flow Errors: A primary block is retired as usual if the predicted address matches with the computed address. If a misprediction occurs, then a flush is initiated for the mispredicted primary blocks and the corresponding redundant blocks. In-

stead, if they match we propose to add the actual and the predicted address both to a block-partitioned Branch Outcome Queue (BOQ). The BOQ will contain eight entries to hold branch outcomes for eight successive blocks. A CRB can retire only if its computed branch address matches both the actual and the predicted address in the corresponding entry of the BOQ, since an error could have occurred either in the primary block branch prediction or in the calculation of the address. A mis-match in either comparison indicates the presence of an error. CRBs do not perform branch prediction, as they use the branch addresses from the primary block stream to access the I-cache.

Detecting Errors in Loads and Stores: The Load Value Queue (LVQ) was proposed for ensuring that the primary and the redundant loads get the same value during redundant execution, even in the presence of intervening stores in multiprocessor systems, which otherwise would lead to false errors [19]. Similarly, the Store Buffer (StB) was proposed to buffer the primary stores temporarily, until they can be compared with the redundant stores, after which they can be committed to permanent storage [19]. Since the StB contains data that is potentially older than the the LSQ, but younger than the caches, the primary loads must access the StB in addition to the LSQ and the caches to compute the return load value.

We augment the TRIPS *load-store queue* LSQ to perform the functionality of the LVQ and the StB, thus eliminating the need for the extra structures and reducing the area overhead. As a brief review, the TRIPS LSQ in the baseline microarchitecture holds all pertinent information relating to in-flight loads and stores. The main functions of the LSQ are to ensure correct ordering between program loads and stores, and detect any load-store dependence violations. The TRIPS LSQ is also statically block-partitioned, and contains eight partitions to support the simultaneous execution of eight primary blocks during normal operation.

We propose a solution that requires no extra LSQ entries, and dynamically divides the existing block partitions allotting up to four partitions to the primary and the redundant blocks at any time. A primary block records its load and store addresses and data in the LSQ

block partition it was allocated at block allocation time. When the primary block completes execution, its LSQ partition is reassigned to the CRB. CRB loads use the data stored by the primary block in the LSQ partition during execution, and match their computed load addresses against the addresses stored by the primary block for fault detection. Similarly, CRB stores compare the computed address and data against the values stored by the primary block for fault detection. An error in either comparison indicates the presence of a fault. The LSQ partition is freed upon successful completion of the CRB, and can be allocated to another primary block. This solution allows the primary block to complete execution and deposit its values in the LSQ from where the CRB can access them later, providing slack between their execution which is attractive both from a performance and a reliability perspective. Similar, to the Store Buffer (StB) described above, primary blocks must also search the LSQ partitions of older primary blocks that were reassigned to CRBs to correctly satisfy program load-store dependences.

Isolation of Register File Errors: The primary blocks and the CRBs use two distinct copies of the register file for fault isolation. With modest extra reconfiguration logic, the redundant blocks can use one of the other four register files provisioned for use in multi-threaded mode. Of course, this would mean that the processor can support fewer threads in redundant execution mode. Since register file errors are isolated, detecting errors in control flow, load addresses, and store addresses and data is sufficient for error detection. This is similar to the larger *Sphere of Replication* proposed in [19]. The CRBs implement full fledged register renaming, and register bypassing similar to primary block execution for higher performance.

System Calls, Exceptions, Timeout: System calls and exceptions act as synchronization points between primary and CRB execution. On a primary block exception, CRB execution is allowed to catch up to that point. If there is no error detected (due to store or branch comparison) until that point, and the CRB also reports the same exception, the exception is

treated as a *true* exception and is reported to the exception handler. On the other hand, if an error is detected before the CRB reaches that point, or if the CRB does not report the same exception, it indicates the presence of a transient fault. System calls are also handled in a similar fashion. If CRB execution reaches the same point without any fault detected, then the system call is serviced. The TRIPS baseline microarchitecture contains logic to raise a timeout exception if a block's lifespan exceeds a pre-determined maximum threshold value. While timeout can occur due to functional errors in the hardware or software, they can also be caused by soft errors that prevent a block from producing all its outputs.

7.1.1 Evaluation of Redundant Execution

As described in Section 6.6, silent data corruption (SDC) is more severe than detected unrecoverable errors (DUE), and the acceptable SDC rate is at least 40X smaller than the target for DUE. For example, IBM targets 114 SDC FIT, and 4566 system-kill DUE FIT for its Power4 systems [2]. Hence, a DUE ACE cycle is at least 40X less malicious than an SDC ACE cycle. Therefore, adding fault detection using redundant execution converts the original SDC ACE cycles into DUE ACE cycles which are at least 40X less malignant. However, while each instruction accumulates SDC ACE cycles once in the original execution, DUE ACE cycles are accumulated both by the original instruction and its redundant copy during redundant execution. Based on this observation, we compute a conservative estimate for the overall ACE cycles for redundant execution by adding the ACE cycles of the primary and the redundant blocks together and derating the total by 40X to convert it to DUE ACE cycles.

As described, while the baseline TRIPS microarchitecture in normal execution mode can execute up to eight speculative blocks, the redundant execution model statically limits the speculation depth of the primary and redundant execution to four blocks. Hence, two factors contribute to the performance overhead: halving the speculation depth and adding redundant execution. Figure 7.3 presents the impact of reducing speculation depth on exe-

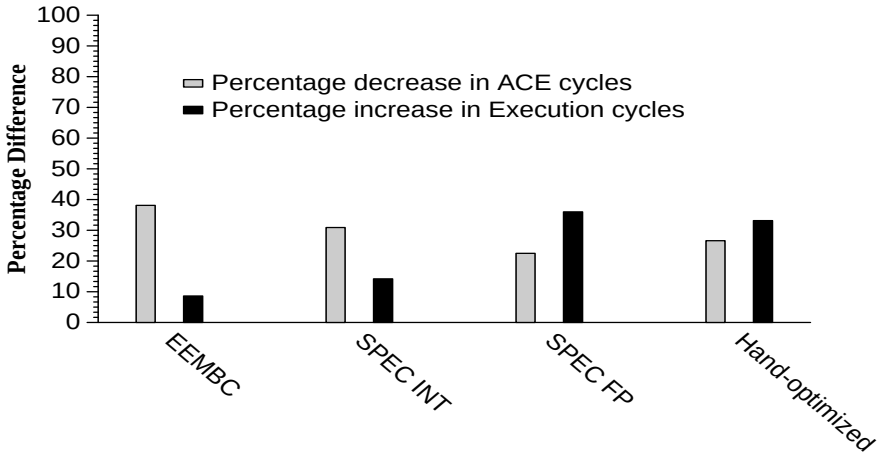

Figure 7.3: Impact on execution cycles and ACE cycles if the maximum number of speculative blocks is reduced from eight to four. The average results are shown for all the four benchmark categories. SPEC floating point benchmarks and hand-optimized benchmarks exhibit high sensitivity to speculation depth.

cution and ACE cycles for each benchmark category. Just restricting program execution to use only four instead of eight blocks, without triggering redundant execution, reduces ACE cycles by 38.1% with 8.6% performance overhead for the EEMBC benchmarks. SPEC integer and floating point benchmarks exhibit 30.9% and 22.5% reduction in ACE cycles, at 14.2% and 36% performance overhead. Similarly, the hand-optimized EEMBC benchmarks show 26.6% decrease in ACE cycles at 33.2% performance overhead when executing using four blocks without redundant execution. SPEC floating point benchmarks and the hand-optimized benchmarks shows higher overhead because of the greater inherent parallelism. In summary, while just halving the speculation depth decreases ACE cycles by reducing the amount of vulnerable state on the processor, it is not a desirable technique for benchmarks with high parallelism because of the poor performance-reliability tradeoff.

Figures 7.4, 7.5, 7.6, and 7.7 present the combined impact of the reduced speculation depth and redundant execution on execution and ACE cycles for the SPEC integer, SPEC floating point, EEMBC, and hand-optimized benchmark categories respectively. Adding redundant execution increases the performance overhead to 20.2% and 49.7%,

while achieving higher reduction in ACE cycles of 83.9% and 87.8% for the SPEC integer and floating point benchmarks. The EEMBC benchmarks exhibit a reduction in ACE cycles by 85.8% at a performance overhead of 20.8%. Finally, the hand-optimized EEMBC benchmarks incur a large overhead of 61.1% to achieve a reduction in ACE cycles of 82.8%. As observed above, the floating point and hand-optimized benchmarks incur the highest overheads due to their greater inherent parallelism. In particular, SPEC floating point benchmarks such as `171.swim` and `172.mgrid`, and hand-optimized benchmarks such as `a2time01`, `basefp01`, and `bezier02` possess a significant amount of parallelism and achieve IPCs between 3-6 in the baseline TRIPS microarchitecture. Hence, they incur a correspondingly high overhead due to the smaller speculation depth, and redundant execution. Mukherjee et al. performed a detailed evaluation of redundant multithreading in an aggressive eight-wide SMT processor resembling the Alpha 21464, and observed an average performance loss of 32% for a suite of SPEC integer and floating point benchmarks [19]. Our results show a slightly higher performance loss of 36% across the 18 SPEC benchmarks. This is not surprising since benchmarks with inherent parallelism exploit the sixteen-wide issue, and the larger instruction window in TRIPS to achieve higher baseline performance, and experience correspondingly higher overhead due to redundant execution.

Sections 7.1.2 and 7.1.3 explore some key optimizations to this baseline redundant execution model to potentially improve the performance overhead. The proposed techniques have so far re-executed the entire primary block to achieve fault detection. We make the observation that the results of primary execution can be used to speed-up redundant execution without sacrificing reliability. In this spirit, we propose two simple techniques to reduce performance overhead without affecting reliability.

7.1.2 Accelerating Dataflow Execution in the Redundant Block

Figure 7.8.a illustrates the current order of events for delivering the redundant load data value; events 1, 2, 3, 4 happen serially. We propose to parallelize this process by decou-

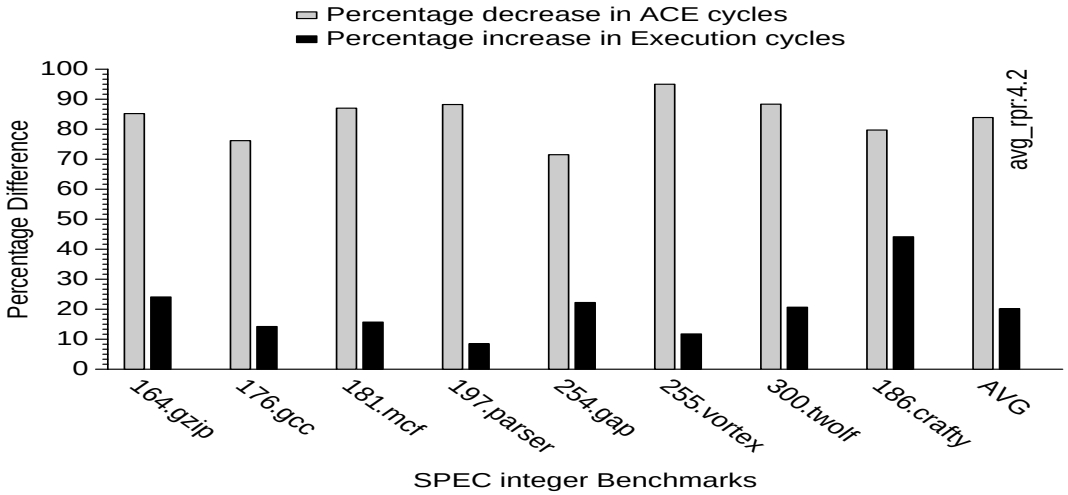

Figure 7.4: Impact on execution and ACE cycles for SPEC integer benchmarks with full redundant execution. The performance overhead is reasonable due to the low inherent parallelism.

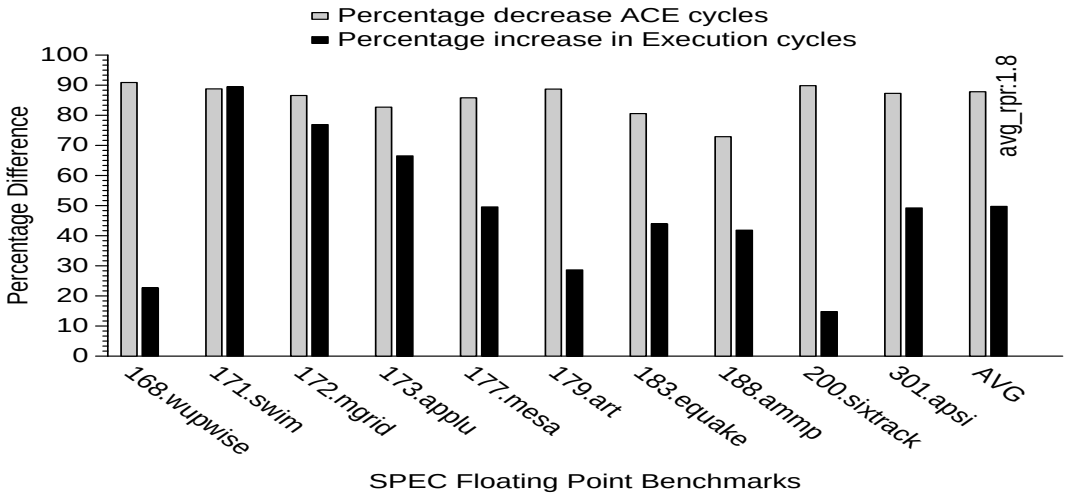

Figure 7.5: Impact on execution and ACE cycles for SPEC floating point benchmarks with full redundant execution. The performance overhead is significant due to the high inherent parallelism.

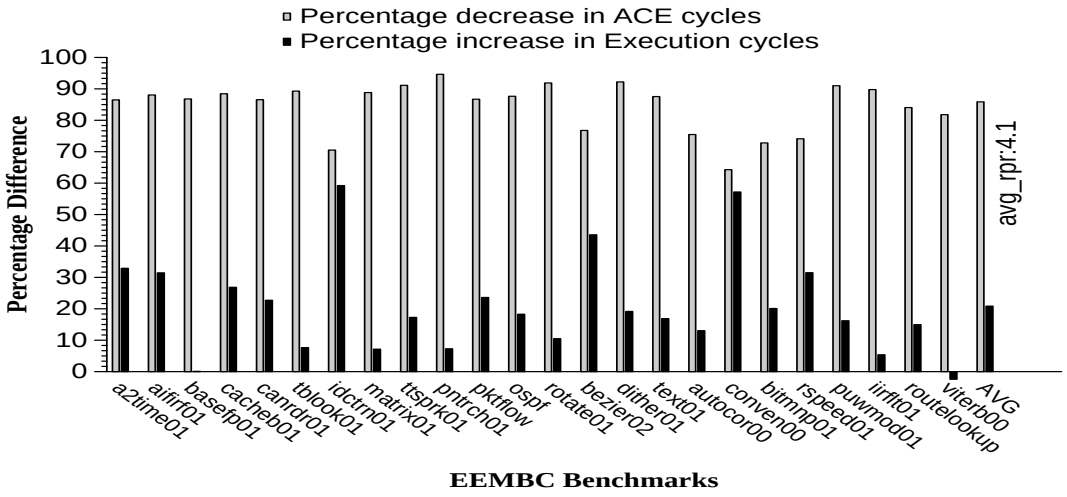

Figure 7.6: Impact on execution and ACE cycles for EEMBC benchmarks with full redundant execution. There is variation in the performance overhead due to the varying degree of parallelism in the benchmarks.

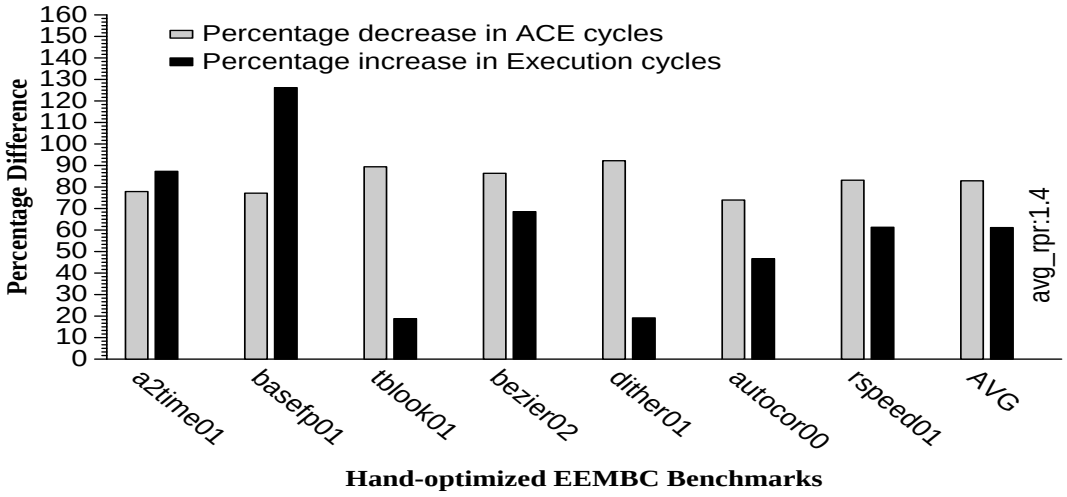

Figure 7.7: Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with full redundant execution. The performance overhead is high because of the high processor utilization exhibited by these benchmarks.

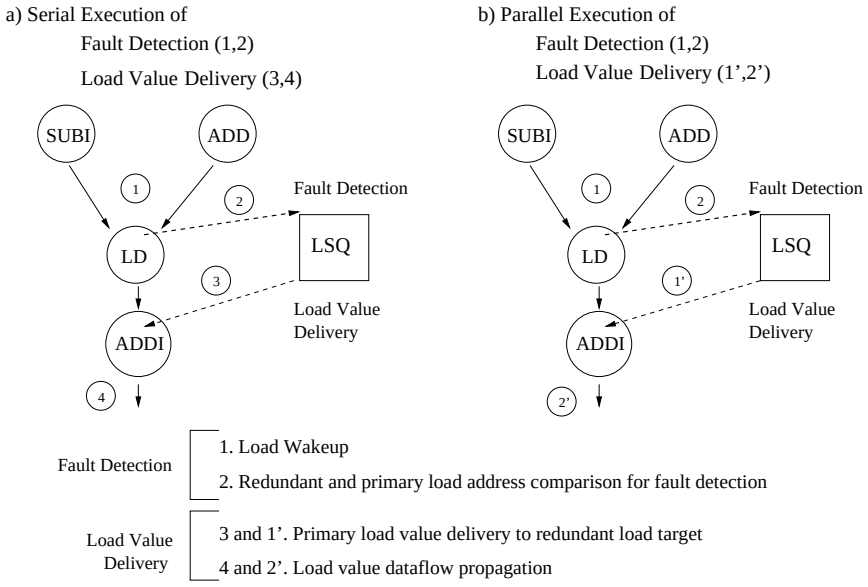

Figure 7.8: Decoupling fault detection and load value delivery in the CRB

pling the load value delivery to the target instruction, and the address comparison for fault detection. Since the instruction physical locations in the TRIPS architecture are known a-priori, the primary block load values residing in the CRB LSQ entries can be delivered to the target instructions as soon as the CRB is mapped. When the redundant load address is computed it is sent to the CRB LSQ entries for comparison. Figure 7.8.b shows the two steps of the decoupled operation, allowing the parallel execution of the two parts of the dependence chain one leading up to the formation of the load address (1 and 2), and the other beginning from the use of the load value (1' and 2'). Austin et al. proposed a similar decoupled mechanism for concurrent value delivery and output comparison to improve the performance overhead of fault detection in conventional superscalar processors [17].

Since value delivery can occur earlier than fault detection due to this optimization as show in Figures 7.8, the block register and store outputs can arrive before fault detection has been completed. Although this normally indicates successful block completion, to achieve fault coverage we must wait for fault detection to complete also. This is ensured by waiting

for exactly as many redundant loads as produced by the corresponding primary block before signalling CRB completion. If there are either greater or fewer loads from the CRB, it indicates the presence of an error.

7.1.3 Predication Model Optimization

TRIPS blocks employ predication at the leaves of the dataflow graph to achieve higher performance by allowing multipath execution within the block, and enabling better utilization of the abundant execution resources [165]. On the other hand, predication at the top of the dependence chains may be more power efficient since it executes fewer instructions. However, for redundant execution we are concerned with the combined performance and power efficiency of both the primary and the redundant blocks. In order to improve this, we propose that the primary block employ predication at the bottom, whereas the CRBs use predication at the top, as shown in Figure 7.9. Redundantly executing fewer instructions saves power, and potentially improves the performance by reducing issue contention.

Since the primary and redundant blocks were identical and fetched from the same address in the baseline redundant execution model, CRB fetch was overlapped with primary block execution to improve performance. Using different predication models in the primary and the redundant block requires them to be stored as separate blocks. To achieve this same overlap between primary block execution and CRB fetch when they are different, the CRB address must be encoded as part of the primary block. Alternatively, a special branch instruction pointing to the CRB can be added to the primary block. This branch instruction will have no dependences and hence can execute immediately, and be interpreted by the GT to trigger the fetch of the CRB. Both these approaches can be equally effective, and while one involves changes to the assembler, and the linker, the other is dependent on at least one spare slot being available for adding the special branch instruction. This technique increases the static size of the program since the CRB's are now different from the primary blocks. The increase in program size can be configured by selectively applying the predication

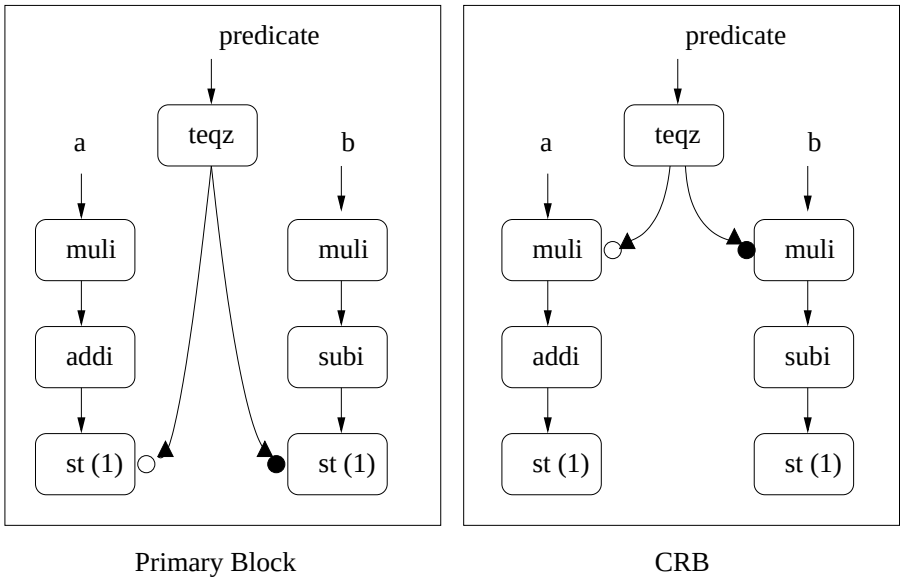

Figure 7.9: Using predication at the bottom in primary blocks for higher performance, and predication at the top in the CRB for redundantly executing fewer instructions.

model optimization to only those blocks that are expected to benefit from it.

7.1.4 Evaluation of Redundant Execution Optimizations

We model these two optimizations to evaluate their benefits. A real implementation of the predication model optimization would require support in the compiler to build primary blocks with predication at the bottom, and redundant blocks with predication at the top. Further, a true implementation would also incur extra memory and cache overhead due to the larger program size. In our simulator implementation of the optimization, we record instructions in the primary block that are not executed either because they are directly or transitively predicated-false, and use this information to invalidate these instructions in the CRB so that they do not contend for resources in any of the pipeline stages in the ET. We use this abstract model, that does not require compiler support and ignores the memory and cache overheads, to measure the potential of this technique. Further, we also evaluate an

extension to this optimization that uses decoupled verification of the predicate values using a predicate value queue (PVQ). Similar to decoupled verification of load values, the CRB must wait until the number of predicate values match those produced by the primary block in the PVQ before signalling block completion.

Figure 7.10 presents the results of the optimizations for the compiler-generated benchmarks. For each benchmark category, the graph presents four sets of results: baseline full redundant execution (FRE), decoupled load value delivery and fault detection (see Section 7.1.2), predication model optimization (see Section 7.1.3), and combined predication model optimization and decoupled predicate value verification. Decoupling load value delivery and verification increases the average overhead in execution time over FRE by 0.4% for EEMBC benchmarks, and provides benefits of 0.75% and 1.75% for the SPEC integer and floating point benchmarks. Since this optimization does not directly reduce contention, and the LVQ already provides a low load-to-use latency for the redundant blocks, it does not provide much benefit. Further, prior research that used decoupled verification and achieved significant reduction in performance overhead applied it to all the redundant instructions and not just the loads [17].

The predication model optimization reduces overhead of FRE by 1.72%, 0.7%, and 1.9% for the EEMBC, SPEC integer, and SPEC floating point benchmarks respectively. Adding the PVQ provides slightly greater benefits of 2.5%, 1%, and 2% for the EEMBC, SPEC integer, and SPEC floating point benchmarks. Adding the PVQ improves the benefit slightly because predicate value delivery and fault detection are done in parallel. Thus, none of the optimizations reduce performance overhead by more than 2.5% for any of the benchmark categories. Our analysis reveals that the predication model optimization is only able to reduce the total number of instructions redundantly executed on the average by 9.1%, 10.8%, and 10.1% for the EEMBC, SPEC integer and floating point benchmarks. As a result, the average number of instructions executed per block reduces by 4.5% for the EEMBC, and by 5.5% for the SPEC benchmarks. Prior research that achieved significant

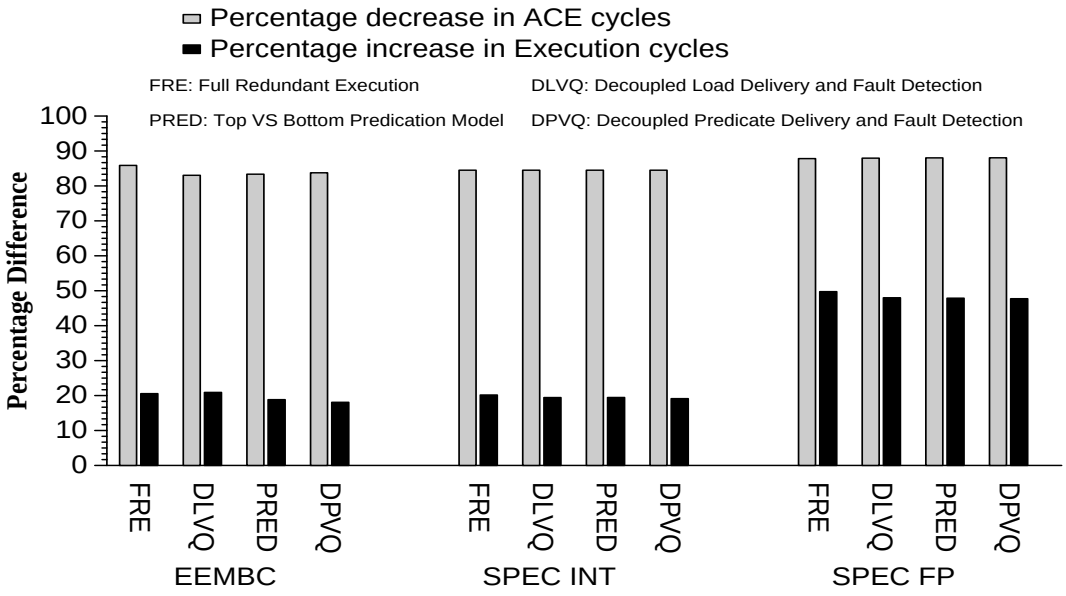

Figure 7.10: Effect of optimizations applied to redundant execution

reduction in performance overhead ($\simeq$10%), used branch and value prediction confidence estimates to eliminate between 30-50% of redundant instructions [117, 118]. While this optimization only filters the small percentage of predicated-false instructions that anyway do not contribute to program outcome, Section 7.3 explores more aggressive mechanisms that trade performance overhead for fault coverage.

7.2 AVF Throttling

We propose AVF throttling as a class of microarchitectural fault evasion techniques that improve soft error reliability by efficiently reducing the amount of vulnerable processor state without significantly extending the execution time. Mukherjee et al. provided the original insight that the cycles that contribute to *architecturally correct execution* (ACE cycles) can be used as an accurate measure of the amount of vulnerable state [123]. As a foundation for studying performance-reliability tradeoff as the interaction between the amount of vulnerable state and the execution time, we present a classification of ACE cycles

into key source components that have a one-one relationship with execution cycles. This classification exposes the reliability bottlenecks in the processor, analogous to execution bottlenecks, and can be used to design the AVF throttling mechanisms.

In general, execution cycles can be divided into three components. Statically determinable cycles including but not limited to functional unit latencies and static communication latencies add up to a certain minimum execution time for the program. Dynamic events such as resource contention and bandwidth constraints build up more latency along program paths and increase the overall program execution time. Slack is the third component, and can be formally defined as the number of cycles an instruction or program path can be delayed without affecting the execution time [159]. Since only the longest path through the dynamic execution of the program or the critical path determines execution time, all paths shorter than the critical path possess slack, and can tolerate greater delay corresponding to the degree of slack. Based on this, we identify three analogous components into which ACE cycles can be broken down into: Min, Slack, and Contention.

Min ACE Cycles: Min ACE cycles is the vulnerable ACE cycles accumulated due to the minimum time useful program state must reside in a structure before it can be used. For instance, if an instruction that contributes to useful program output reads data from the register file, the data must have resided there for at least one cycle before it is read. This cycle during which time it is vulnerable to an error, contributes to the Min ACE cycles of the program. However, it is possible that the instruction does not read its operand from the register file, and instead receives it from the operand bypass network. Of course, in this case the register file does not accumulate any ACE cycles due to this operand. Min ACE cycles can be computed for many structures in conventional processors including the physical register file, the instruction window and the reorder buffer. It also naturally applies to TRIPS processor structures such as the register file, read and write queues, instruction and operand buffer, LSQ structures, and the operand router buffers.

Slack ACE Cycles: While slack cycles do not affect the execution time of the program, the useful processor state associated with that slack accumulates ACE cycles during this period. Fields et al. identified two primary flavors of slack: local and global, each with different scope. **Local slack** of a dynamic instruction *i*, is the number of cycles it can be delayed without delaying any subsequent instruction. They analyzed the SPEC 2000 benchmarks, and showed that approximately 20% of the instructions have local slack greater than five cycles. **Global slack** of a dynamic instruction *i*, which is more aggressive, is the number of cycles it can be delayed without extending the execution time of the program. They showed they approximately 40% of instructions have global slack greater than 50 cycles. Local slack is easier to analyze than global slack, since analyzing global slack typically needs a critical path model of the whole program [159]. In this dissertation, we focus on studying the interaction between local execution slack and ACE cycles, but we recognize that global slack can also have a substantial impact on ACE cycles and is worth exploring in the future.

There are numerous scenarios where *local slack* ACE cycles are accumulated in both conventional superscalar processors, and distributed architectures such as the TRIPS processor. Slack ACE cycles are accumulated when a `read`, `write`, or a normal instruction arrives ahead of its operand, and waits in the read queue, write queue, or in the instruction buffer until it is ready to be executed. Of course, the entry in the write queue, read queue, or operand buffer that stores the operand accumulates slack ACE cycles, if the operand arrives ahead of the instruction. Techniques such as front-end throttling during periods of low utilization may be effective in reducing these slack ACE cycles. Similarly, a register value that is available in the register file, or in the write queue, before it is read by an instruction accumulates slack ACE cycles in that duration. Compiler or hardware instruction scheduling techniques can delay register writes and advance register reads to compress the vulnerable window, and reduce slack ACE cycles in the register file and write queue. Architectural constraints such as in-order commit of block register writes from the

write queue, and stores from the LSQ can also force younger blocks to accumulate slack ACE cycles while waiting for older blocks to commit. In all these examples, the ACE cycles accumulated over and above the minimum are grouped under slack ACE cycles. However, there are structures such as the OPN router where packets in the buffer have no local slack, and are ready to leave as soon as they enter. Hence packet occupancy in the buffer contributes only to Min, and Contention ACE cycles which is described next. As mentioned before, the contribution to slack ACE cycles will be different and will increase if we take into account global execution slack also.

Contention ACE Cycles: Resource contention can delay events that are already ready to trigger, and the extra cycles incurred due to contention are classified under Contention ACE cycles. Using the same example, an instruction which has received all its operands and is ready for execution, but is not issued for execution due to contention at the instruction selection stage, accumulates Contention ACE cycles during that period. Similar opportunities for Contention ACE cycles exist in each ET that implements single instruction issue per cycle, in each RT that may issue exactly one read and one write instruction every cycle, in each DT that can support up to one load and store access every cycle, and in each OPN router that can send up to one control and data packet in each direction. In all of these cases there is potential for resource contention and adding Contention ACE cycles, when the number of contenders exceeds the supported bandwidth.

Baseline Distribution of ACE Cycles: Figure 7.11 first presents the breakdown of each component of ACE cycles into the relative contributions from the individual hardware structures. These results represent the average across all the four benchmark categories. The read and write queues, OPN buffers, and the instruction buffers, in that order, are the top three contributors to the Min ACE cycles. Contention ACE cycles are distributed more evenly with the OPN buffers contributing the maximum. Finally, the instruction buffers account for about 70% of the Slack ACE cycles, dominating that component.

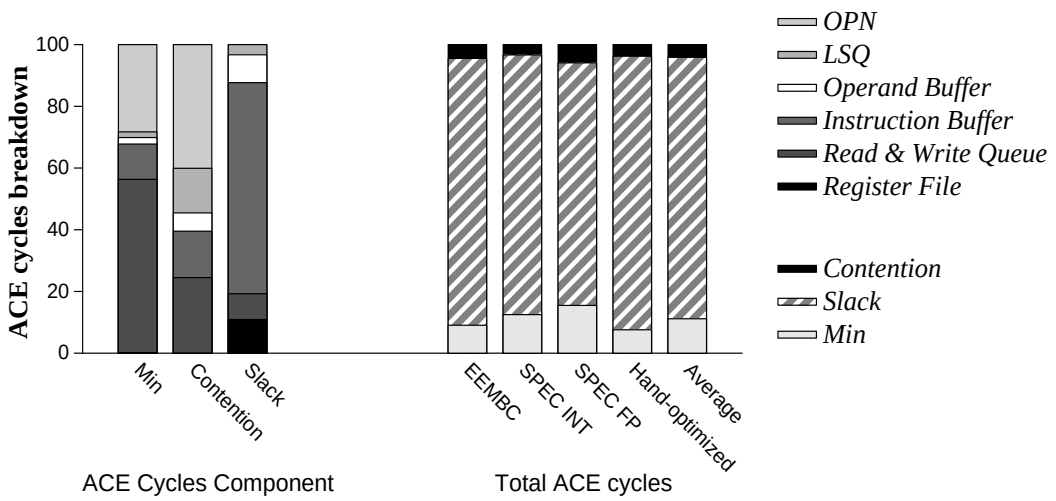

Figure 7.11: Graph presents the breakdown of each ACE cycle component: Min, Slack, and Contention among the different processor structures. These results present the average across all the benchmark categories. Graph also shows the breakdown of the total ACE cycles among the three components: Min, Slack, and Contention for each benchmark category.

Figure 7.11 also presents the relative contribution of the three components: Min, Slack, and Contention to the total ACE cycles. The graph clearly indicates that slack constitutes between 80-90% of the total ACE cycles for all the benchmark categories. The dominant contribution from Slack ACE cycles emphasizes the nature of ACE cycles as a cumulative property that is accumulated across all paths of program execution, unlike execution cycles which is determined only by the length of the longest path through the dynamic program execution. On the average across all the benchmark categories, Min ACE cycles contribute 11.1%, Slack ACE cycles contribute 84.7%, and Contention ACE cycles contribute 4.2% to the total ACE cycles. The results in Table 6.6 showed that the instruction window contributes 63.5% to the total ACE cycles across all the three components (Min, Slack, and Contention). Based on the analysis here, we find that a dominant fraction of 59% (70% x 84.7%) of this contribution comes just from Slack ACE cycles.

Our result corroborates prior research using more conventional processors, which have also demonstrated the high vulnerability of the instruction window [123]. The large

1024 entry instruction window, together with the block-atomic execution model in the TRIPS architecture further increases its vulnerability. Unlike a conventional processor, the block-atomic execution model fetches all the instructions of the block together, taking into account the delay due to pipelining, and hence instructions, especially those at the bottom of the dataflow graph, are fetched much ahead of their operands substantially increasing the *local* slack ACE cycles. Further, since eight such blocks can be speculatively mapped on the large instruction window, block instruction slack ACE cycles increases with speculation depth (blocks lower in the control flow graph), because their execution may be delayed due to control and/or data dependences with the older blocks.

Based on these results, we propose and evaluate AVF throttling techniques focused on reducing the slack ACE cycles in the instruction window. The techniques exploit slack to defer instruction fetch and reduce the amount of vulnerable state without extending execution time. There have been several prior techniques proposed to estimate slack:

Delay and observe: A simple approach that estimates instruction slack by delaying its execution by n cycles, and observing if the overall execution time is affected. Srinivasan et al. used this approach to compute the latency tolerance of program loads [166].

Static slack estimation: Compilers generally use static estimates of instruction slack to perform efficient instruction scheduling within region boundaries (e.g., hyperblock) to achieve higher performance [115, 167].

Profiling using a dependence-graph model: Fields et al. proposed an off-line profiling methodology based on constructing a dependence-graph model of dynamic program execution and analyzing the graph to determine execution slack [159]. Muthler et al. estimated slack using this methodology and used it to improve performance [161].

Dynamic slack prediction: Fields et al. also implemented a dynamic hardware slack prediction mechanism to achieve processor energy reduction [159].

In this study, we explore a modified version of the first two techniques for reducing instruction slack, and thus the Slack ACE cycles, while recognizing that the remaining two techniques are also equally applicable. The first technique operates at the inter-block granularity and attempts to spread the fetch and execution of successive speculative blocks to reduce slack ACE cycles. On the other hand, the second technique functions at the intra-block level, and orchestrates the relative timing of block instruction fetch and block execution to lower the instruction slack ACE cycles. The two techniques are complementary since they operate at different granularities (intra- and inter-block), and can naturally be used together to achieve greater benefits.

7.2.1 Dynamic Speculation Control (DSC)

Speculative execution is a mainstream technique used to exploit concurrency in high performance microprocessors. The efficiency of speculative execution depends on the inherent parallelism available in the program and the capacity to correctly predict the path that will be taken by the program in the future. Even if the processor succeeds in correctly predicting the execution control flow, speculation can be very inefficient if there is little inherent parallelism in the application. In this case, while the application sees very little performance improvement, program state is unnecessarily brought into the processes much before it is required degrading the reliability of the data when it is actually used.

Figure 7.12 shows the incremental impact of instruction window size on execution and ACE cycles, averaged across the compiler-generated EEMBC benchmarks. For instance, while the first set of bars measure the relative change in ACE and execution cycles going from 128 to 256 instructions, the second set considers 256 instructions as the baseline and measures the relative change going from 256 to 512 instructions. While execution time shows diminishing returns of 38%, 23% and 8% as window size increases, ACE cy-

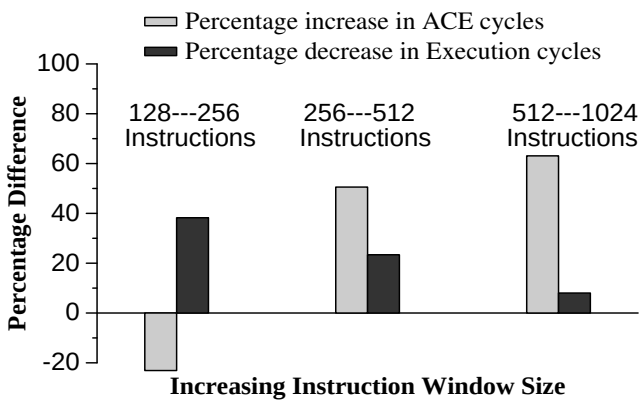

Figure 7.12: Impact of speculation depth and increasing instruction window size on execution and ACE cycles. Graph shows that while execution cycles shows diminishing returns, ACE cycles exhibits accelerated growth with increasing window size.

cles exhibits the opposite trend of accelerated growth going from -23% to 50% to 63%. Increasing the instruction window from 128 to 256 instructions benefits both performance and reliability by 23% and 38% respectively. Programs are able to use the larger window to achieve higher parallelism and and hence reduce execution time, which inturn reduces the total ACE cycles accumulated. However, at higher window sizes the smaller execution time is outweighed by the accelerated increase in the amount of processor state leading to an overall increase in ACE cycles. Based on this result, we propose a dynamic technique that uses an adaptive algorithm to determine the optimum size of the instruction window for each program interval. Dynamically exploring different discrete sizes of the instruction window to determine the optimal capacity is similar in concept to the delay and observe scheme. The instruction fetch engine will be throttled in a particular cycle if the instruction window reaches this threshold, to maintain this optimal capacity. We first describe the adaptive algorithm before presenting the results.

The algorithm basically uses gradient descent to arrive at the optimal setting. The execution of a program is divided into multiple intervals. The IPC and the ACE cycles accumulated during each execution interval are computed using two hardware performance counters. Currently, we use the simulator to measure the ACE cycles in each interval, as

described in Section 6.5. We assume that in a real implementation, ACE cycles can be measured accurately using hardware performance counters. Using performance counters to measure ACE cycles was also suggested in prior work [123], and we think it is a promising area for future work. The first interval is used to determine the baseline performance in instructions-per-cycle (IPC), and ACE cycles for the program in that interval. In subsequent intervals, the instruction window size is varied in pre-determined discrete amounts and the resulting IPC and ACE cycles are measured for each new setting. The RPR for each window size is also measured as the ratio of the percentage improvement in ACE cycles to the percentage drop in IPC (as described in Section 6.3). To be eligible, a new instruction window setting must meet two conditions: 1) the performance overhead should be within a specified threshold, and 2) the RPR should meet or exceed the input RPR specification. The optimal setting is the window size that meets these two conditions and maximizes RPR. The window size is left at the baseline capacity if there is no setting that meets these conditions, although there is still some performance overhead due to this sampling process. The program then continues execution with this optimal setting for the next 20 intervals, after which this entire process is repeated to account for changes in program behavior.

We chose the interval size to be 50K cycles to react quickly to changes in program behavior without letting the tuning mechanism cause a high overhead. The baseline TRIPS architecture has a distributed instruction window that can accommodate a total of 1024 instructions. The algorithm is allowed to examine four different instruction window sizes: 1024, 768, 512 and 256 instructions, corresponding to 8, 6, 4, and 2 speculative TRIPS blocks, and the algorithm chooses a size dynamically. The algorithm was constrained to be within a maximum performance overhead of 15%, and meet an RPR of 1 ie. the benefit in reliability should at least be comparable to the performance overhead.

Figures 7.13, 7.14, 7.15, and 7.16 present the results of this scheme for the SPEC integer, SPEC floating point, EEMBC, and hand-optimized benchmarks respectively. Again, these results are with respect to the baseline TRIPS processor operating in normal exe-

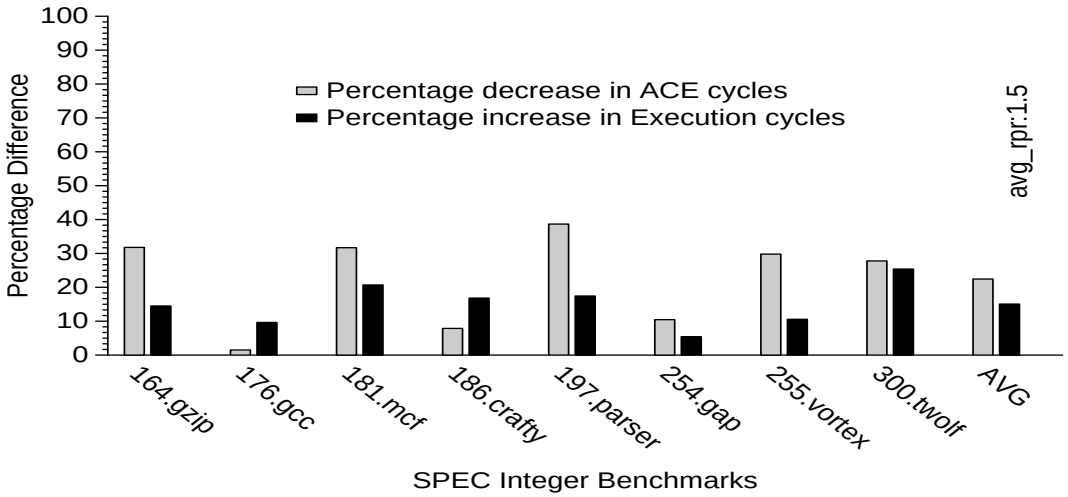

Figure 7.13: Impact on execution and ACE cycles for SPEC integer benchmarks with dynamic speculation control.

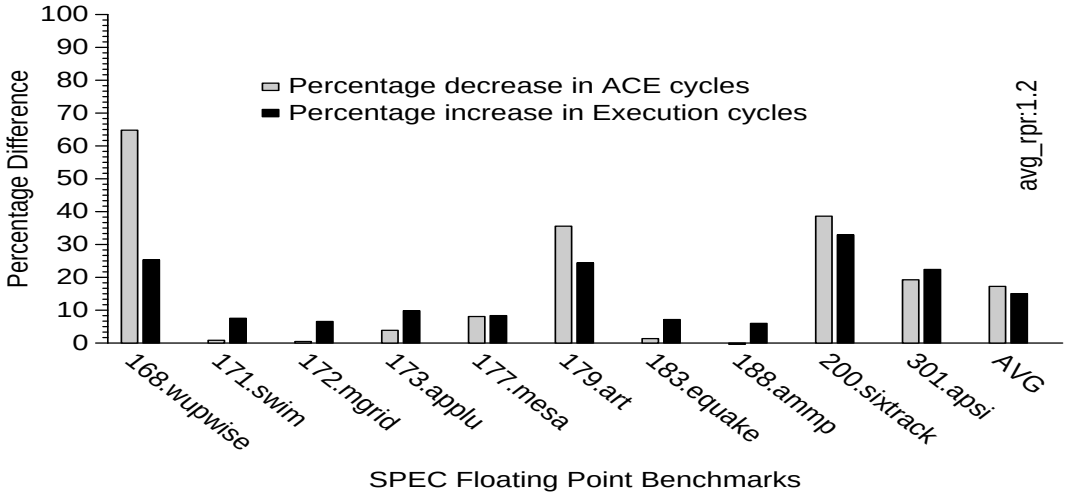

Figure 7.14: Impact on execution and ACE cycles for SPEC floating point benchmarks with dynamic speculation control.

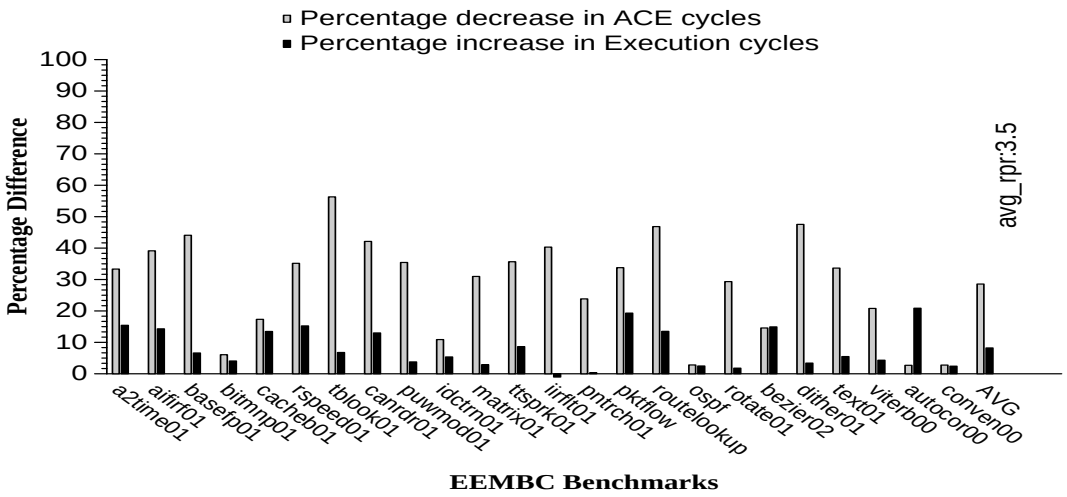

Figure 7.15: Impact on execution and ACE cycles for EEMBC benchmarks with dynamic speculation control.

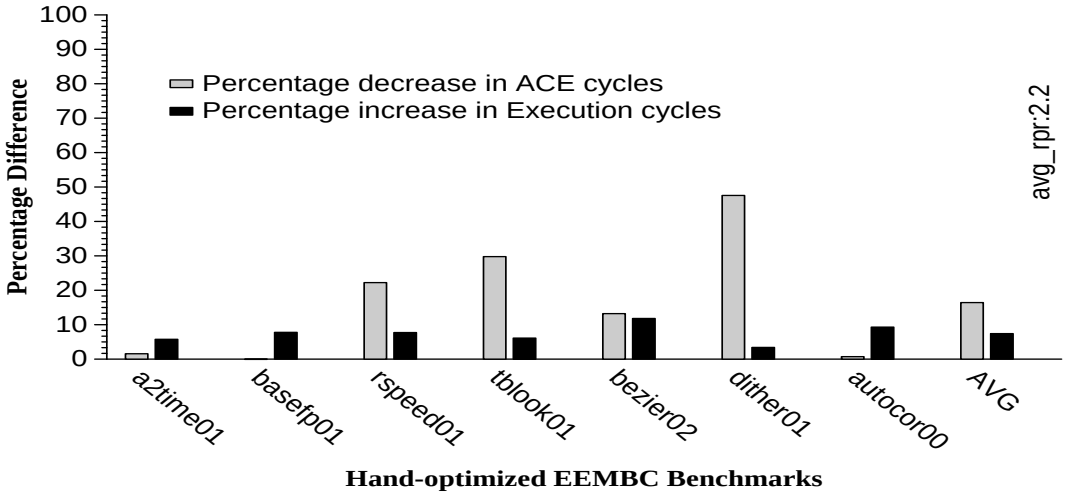

Figure 7.16: Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with dynamic speculation control.

cution mode using all the resources. The technique is reasonably effective for the SPEC integer benchmarks reducing ACE cycles by 22.5% at a performance overhead of 15% resulting in an RPR of 1.5. Across the ten floating point benchmarks, the technique achieves an improvement of 17.25% in ACE cycles at a cost of 15% in execution time, with only `wupwise`, `mesa`, `art`, `sixtrack`, and `apsi` showing real benefits. Since the majority of SPEC floating point benchmarks are structured as tight loops with high inherent parallelism and good predictability, they can very effectively exploit speculative execution and are very sensitive to instruction window capacity. The technique is very effective for the EEMBC benchmarks, and reduces ACE cycles by 28.6%, at a cost of 8.2% in execution cycles, with an average RPR of 3.5. On the other hand, the technique is least effective for hand-optimized benchmarks achieving only 16.4% reduction in ACE cycles at 7.4% performance overhead. As expected, the technique performs less well for the floating point and hand-optimized benchmarks which use the speculation depth effectively. Overall, it emphasizes the potential of efficient speculation in improving soft error reliability.

7.2.2 Fetch-on-Demand (F-o-D)

Dynamic speculation control trades performance for reliability in a coarse grained fashion by placing hard limits on instruction window size. The goal of fetch-on-demand (F-O-D) is to further reduce the performance overhead by allowing the full window capacity, and achieve reliability improvement by exploiting instruction slack to trigger instruction fetch just when it is required, thus minimizing the slack ACE cycles in the instruction window. As described in Section 4.2.5, instruction fetch for the entire block is triggered immediately upon block allocation, leading to the high slack ACE cycles observed in Figure 7.11. We build fetch-on-demand using the simple observation that execution of the current instruction block effectively commences when its data dependences with older blocks in the program are satisfied through the data forwarding logic. In particular, we use the inter-block register dependences, delivered through the register forwarding logic from one of the other

concurrently executing blocks, as a natural indicator for the beginning of block execution. The goal here is to overlap the time the register value takes to travel from the register file interface to the consumer instruction at the destination execution unit, with the time the instruction takes to travel from the instruction cache interface to the same execution unit, to minimize instruction slack.

We first perform a preliminary analysis to explore the potential of this observation, based on which we design a specific technique. We examine two fixed choices for register inputs for triggering block instruction fetch. The first trigger that we investigate, is the arrival of the first register input to the instruction block through the register forwarding logic in the write and the read queues, from one of the other concurrently executing blocks. If all of the register inputs are already present in the register file, block fetch is triggered only after sending all the register inputs. The second scheme we examine is to trigger block instruction fetch only after sending all the block register inputs, regardless of whether they are available from the register file or the register forwarding logic. While the baseline TRIPS processor pipelines the instruction fetch of multiple blocks so that they are sent back to back if they are available, we augment the fetch logic to begin the block fetch only on the occurrence of the trigger event.

We performed the preliminary evaluation for all the benchmark categories. Figure 7.17 presents the summary results for the first trigger, fetch-on-demand on arrival of the first register input on the left; and the second trigger, fetch-on-demand on sending all the block register inputs on the right. Triggering instruction fetch on arrival of the first register input performs well for all the four benchmark categories. EEMBC benchmarks show 28.2% benefit in ACE cycles at 4.4% overhead, SPEC integer benchmarks provide 29.4% benefit in ACE cycles at 3.5% overhead, SPEC floating point benchmarks exhibit 10.7% benefit in ACE cycles at 1.1% overhead, and hand-optimized benchmarks show 10.7% reduction in ACE cycles at 4.5% overhead, with RPR significantly greater in comparison with dynamic speculation control for all the four categories. Thus, fetch-on-demand clearly

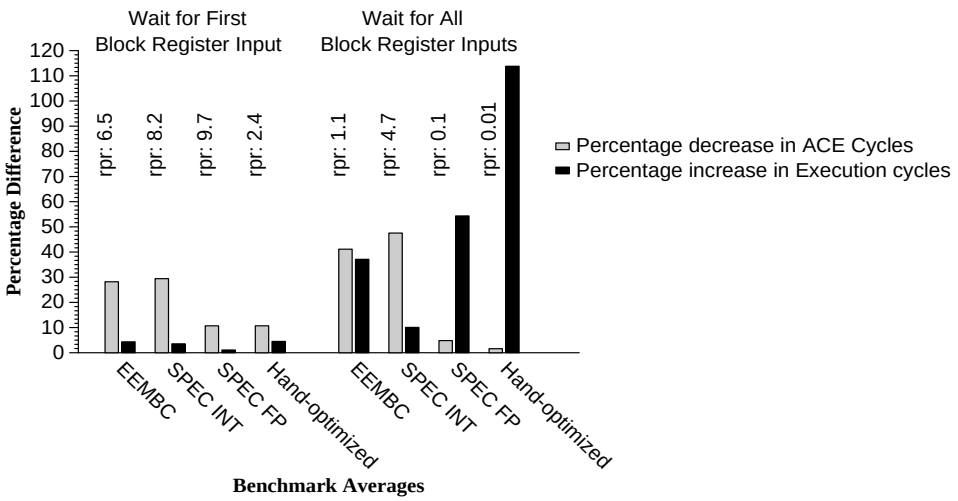

Figure 7.17: Exploration of potential performance-reliability tradeoff with fetch-on-demand.

has greater potential than dynamic speculation control as it provides comparable reliability improvement, at much smaller performance overhead.

Triggering instruction fetch only after sending all the block register inputs shows dramatically different results. Interestingly each of the four benchmark categories exhibit different behavior. EEMBC benchmarks show greater reliability improvement of 41% at a correspondingly high overhead of 37%. On the other hand, SPEC integer benchmarks achieve a favorable tradeoff with a larger 47.5% reduction in ACE cycles at only 10.1% overhead, exposing the lower inherent parallelism due to data dependences in these programs. SPEC floating point benchmarks which have very high parallelism expectedly show the opposite trend with a reduction in ACE cycles of only 4.8% at a large performance cost of 54.5%. The worst performance-reliability tradeoff is exhibited by the hand-optimized benchmarks which achieve a reduction in ACE cycles of only 1.6% at an unacceptable performance overhead of 113%. In summary, this preliminary evaluation demonstrates that substantial savings in ACE cycles can be obtained, if the processor intelligently uses specific microarchitectural events as triggers to expose execution slack. Based on this result, we explore a compiler-based technique for implementing fetch-on-demand.

Compiler-based Static Slack Estimation

Prior research has shown that estimating global slack is quite complicated and requires a detailed critical path model [168,169]. In this dissertation, we limit our analysis and evaluation to local execution slack to simplify the problem, but we recognize that global slack can also have a substantial impact on ACE cycles and is worth exploring in the future. Further, prior work has also demonstrated that exploiting slack dynamically, and in a timely fashion, requires complex hardware slack predictors [159]. We make the key observation that the block-atomic execution model provides an inherent advantage, as the existence of slack can be detected at the block boundary allowing enough time for it to be exploited during the lifetime of the execution of the block. Further, we propose that the instruction scheduler (described in Section 4.2.2), which has detailed models for the critical path and the slack associated with each primary block register input, can be used for estimating slack at the block boundary, without using hardware slack predictors. As described in Section 4.2.2, the TRIPS compiler performs instruction scheduling for each instruction block. While the scheduling region is limited to a single block, the algorithm also accounts for inter-block (global) critical paths (such as loop-carried dependences) to model the block register input arrival times, which is critical in estimating the slack of register inputs [115].

Based on these observations, we propose a compiler-based static estimation of slack associated with each block register input, to identify a specific block register input trigger to maximize reduction in ACE cycles, and minimize execution time overhead. The technique identifies the appropriate block register input to trigger just-in-time instruction fetch of the block, holding the instructions until that time in ECC or parity protected instruction caches. If the register input trigger is too early then there is not much reduction in the slack ACE cycles. On the other hand, if the register input trigger is too late, it can significantly extend execution time, and hence also lead to an overall increase in ACE cycles.

At the end of scheduling the block, the same algorithm is used to compute the critical path length and slack of each block register input. The register input with the longest

critical path length has no slack, and if used to trigger block fetch will most likely extend the execution time. As described earlier, the goal is to overlap the time the register value takes from the register file interface to the consumer instruction, with the time the instruction takes from the instruction cache interface. The compiler approximates this by annotating each block with the register input that has an estimated arrival time just smaller than that of the register input with the longest critical path length. The TRIPS hardware records the register input annotation for each block, observes the OPN for the arrival of the specific register input, and upon its arrival takes advantage of the independent control over the different on-chip networks to trigger instruction fetch for the block on the GDN.

Figures 7.18, 7.19, 7.20, and 7.21 present the results of this scheme for the SPEC integer, SPEC floating point, EEMBC, and hand-optimized benchmarks respectively. Again, these results are with respect to the baseline TRIPS processor operating in normal execution mode using all the resources. To summarize the results, the technique is very effective for the SPEC integer benchmarks, and reduces ACE cycles by 30%, at a cost of only 4.5% in execution time, with an average RPR of 6.7. The floating point benchmarks are again difficult, and the technique only achieves an improvement of 9% in ACE cycles at a cost of 11.2% in execution time, with only `wupwise`, `ammp`, `equake`, and `sixtrack` showing real benefits. For the EEMBC benchmarks, the technique is able to reduce ACE cycles by 27.5% with 7.8% overhead in execution time. It performs poorly for the hand-optimized EEMBC benchmarks and reduces the ACE cycles only by 3.6% but incurs 15.8% performance overhead. On the average, the technique performs better or at least as well as dynamic speculation control for the SPEC integer and EEMBC benchmarks. On the other hand, it is less effective than dynamic speculation control for the SPEC floating point and hand-optimized benchmarks which have lesser slack available.

An interesting thing to note is that, some benchmarks including the `a2time01` and `dither01` hand-optimized benchmarks, and `swim` and `apsi` SPEC floating point benchmarks actually exhibit an overall increase in ACE cycles. In these cases, the technique

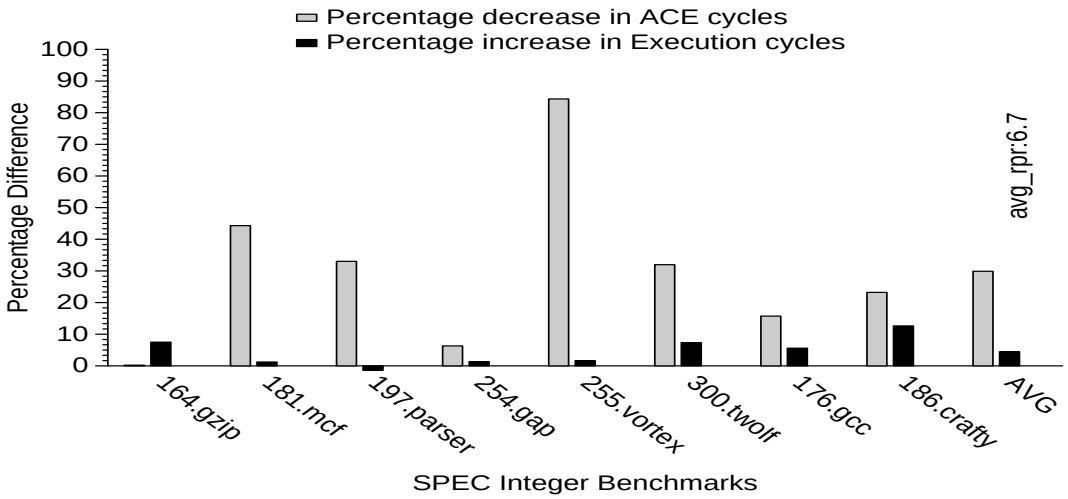

Figure 7.18: Impact on execution and ACE cycles for SPEC integer benchmarks with fetch-on-demand using static slack estimation.

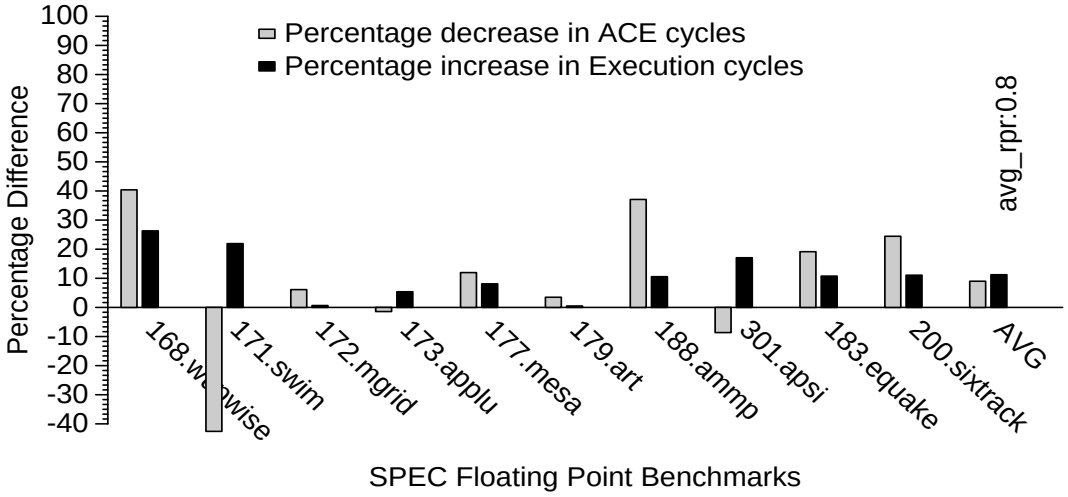

Figure 7.19: Impact on execution and ACE cycles for SPEC floating point benchmarks with fetch-on-demand using static slack estimation.

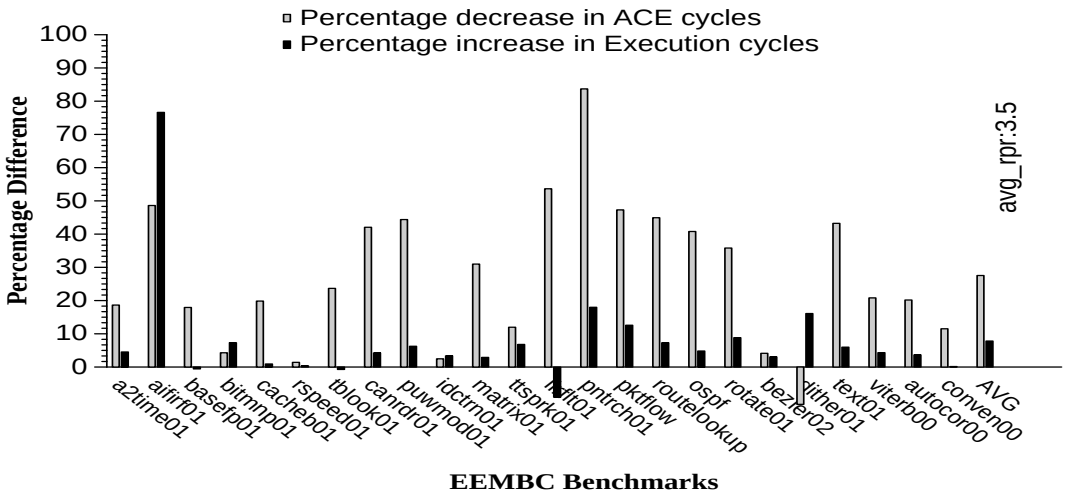

Figure 7.20: Impact on execution and ACE cycles for EEMBC benchmarks with fetch-on-demand using static slack estimation.

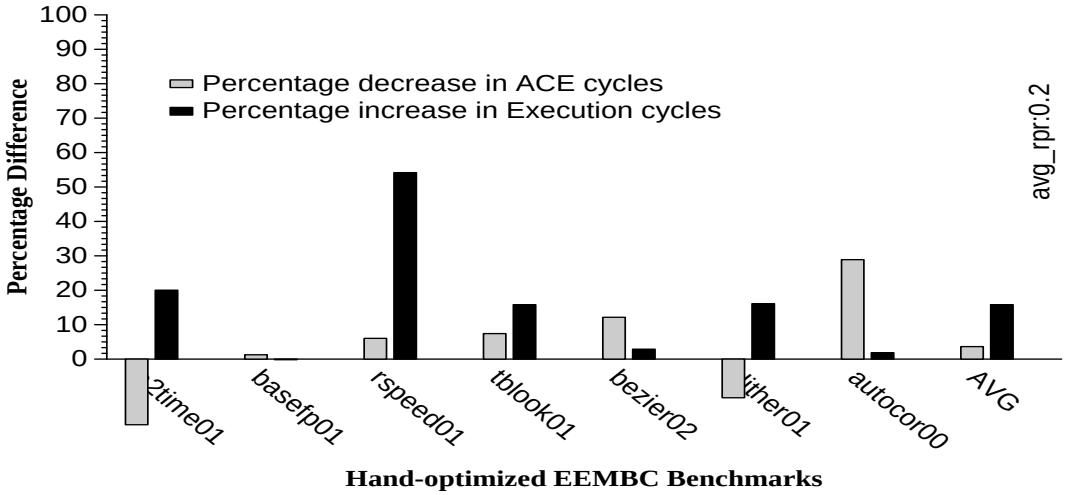

Figure 7.21: Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with fetch-on-demand using static slack estimation.

over-estimates the slack, and the resulting increase in execution time outweighs the benefits, and leads to an overall increase in ACE cycles. Similarly, the hand-optimized benchmark `rspeed01` also shows a large increase in execution time relative to the reduction in ACE cycles, again indicating that the technique over-estimates the slack for this benchmark. While the floating point and hand-optimized benchmarks have less slack and thus require more accurate slack estimation in general, we identify two primary causes for this over-estimation. First, while the technique is attractive for its simplicity in using the compiler for estimating slack, the lack of dynamic knowledge limits its accuracy. Combining static slack estimation with dynamic feedback about the quality of the static decision, similar to delay and observe, can compensate for the over-estimation. Second, the technique in its present form only considers the local slack of program paths within a block originating at a block register input. Although the scheduler accounts for global critical paths in a limited form, detailed analysis of global slack can potentially improve the accuracy of this technique albeit at higher complexity.

7.2.3 Comparison of Dynamic Speculation Control and Fetch-on-Demand

Figure 7.22 presents the comparison between the two AVF throttling techniques. For each benchmark category, the graph first shows the reliability improvement and performance overhead from using each of the techniques individually and then in combination. Since the two techniques operate at complementary intra- and inter-block granularities, combining them promises even higher benefits. Further, the results in Sections 7.2.1 and 7.2.2 show that for some benchmarks such as `art` and `ammp` one technique performs better than the other, suggesting that using them together can achieve improvements for both. Overall, the fetch-on-demand performs better or at least as well as dynamic speculation control for the SPEC integer and EEMBC benchmarks. On the other hand, it is less effective than dynamic speculation control for the SPEC floating point and hand-optimized benchmarks which have lesser slack available. In combination, they provide greater average reduction in

ACE cycles than either of them used individually, but at higher performance overhead. This comparison for the individual benchmark categories is analyzed in greater detail below.

Figures 7.23, 7.24, and 7.25 present the results for SPEC, EEMBC, and the hand-optimized benchmarks when the two techniques are used together. In summary, SPEC integer and floating point benchmarks show ACE cycles reduction of 37.5% and 25.7%, with execution time overhead of 15% and 25.8% respectively. Further, both `art` and `ammp` now show a considerable decrease in ACE cycles. EEMBC benchmarks display ACE cycles reduction of 42.7% at a higher execution time overhead of 16.8%. Finally, the hand-optimized benchmarks achieve 23% reduction in ACE cycles at 24% performance overhead.

Overall, the results show greater reduction in ACE cycles from combining the techniques, improving over both dynamic speculation control and fetch-on-demand for all the benchmark categories, albeit at a higher performance overhead. In particular, for the hand-optimized benchmarks `tblook01`, `rspeed01`, and `bezier02` the combination achieves greater reduction in ACE cycles by accepting a higher performance overhead. On the other hand, while the combination incurs higher overhead it achieves less improvement in reliability than just using dynamic speculation control for the hand-optimized benchmark `dither01`. Similarly, fetch-on-demand performs better in isolation for `autocor00` than the combination. This emphasizes the differences in benchmark characteristics; while some are very sensitive to the degree of speculation they have slack within the block, and others have little slack within blocks but do not use the full speculation depth effectively.

7.2.4 Measuring Actual Reduction in Instruction Slack ACE Cycles

Although these techniques were motivated by the dominant contribution of the local slack ACE cycles in the instruction window (see Figure 7.11), the overall ACE cycles reduction achieved by each technique accounts for the cumulative effect in ACE cycles observed across all of the TRIPS processor structures tracked in Table 6.1. Figure 7.26 measures how effective the techniques are in reducing the slack ACE cycles in the instruction window. For

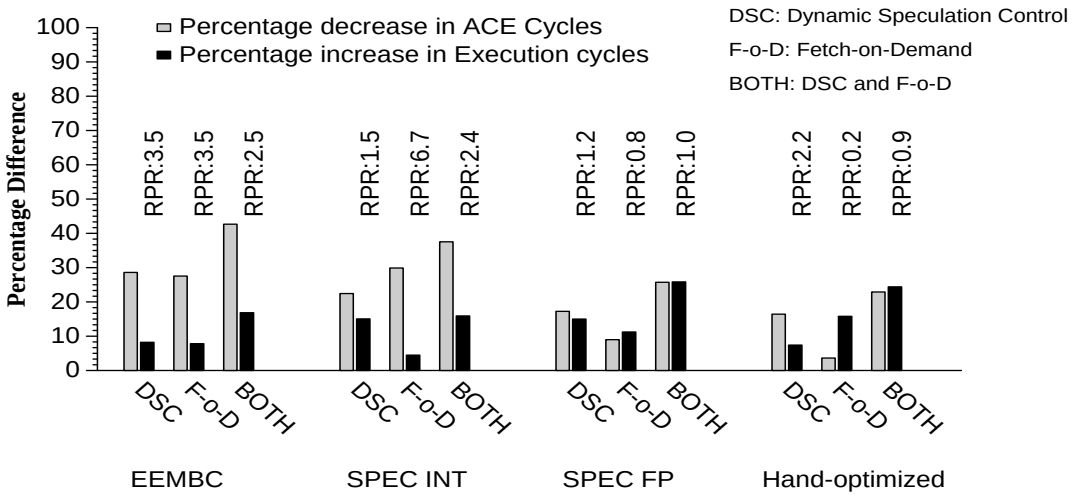

Figure 7.22: Comparison of AVF throttling mechanisms for all the benchmark categories. Graph confirms that the combination of the two techniques provide greater benefits that the benefits from using each of them individually.

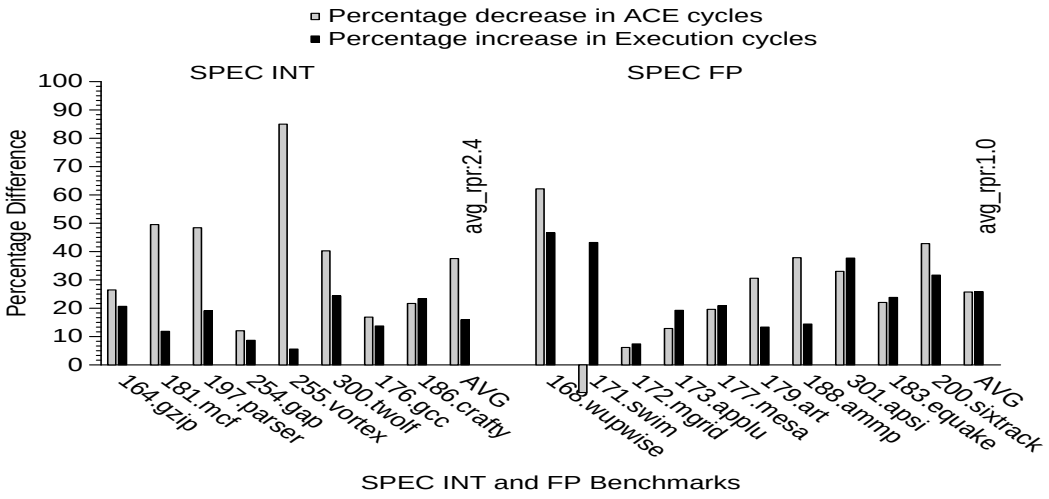

Figure 7.23: Impact on execution and ACE cycles for SPEC benchmarks with fetch-on-demand and dynamic speculation control. Graph shows the potential benefits of AVF throttling.

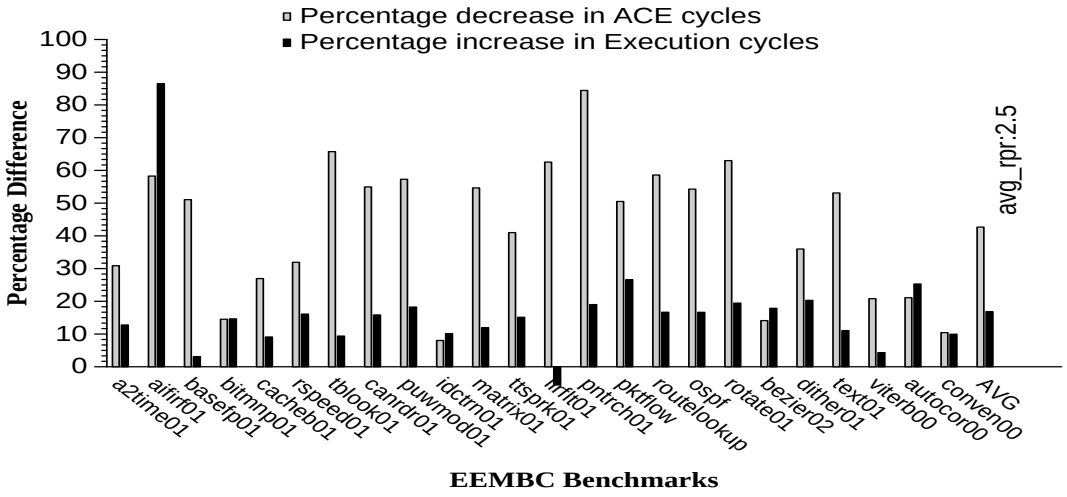

Figure 7.24: Impact on execution and ACE cycles for EEMBC benchmarks with fetch-on-demand and dynamic speculation control. Graph shows the potential benefits of AVF throttling.

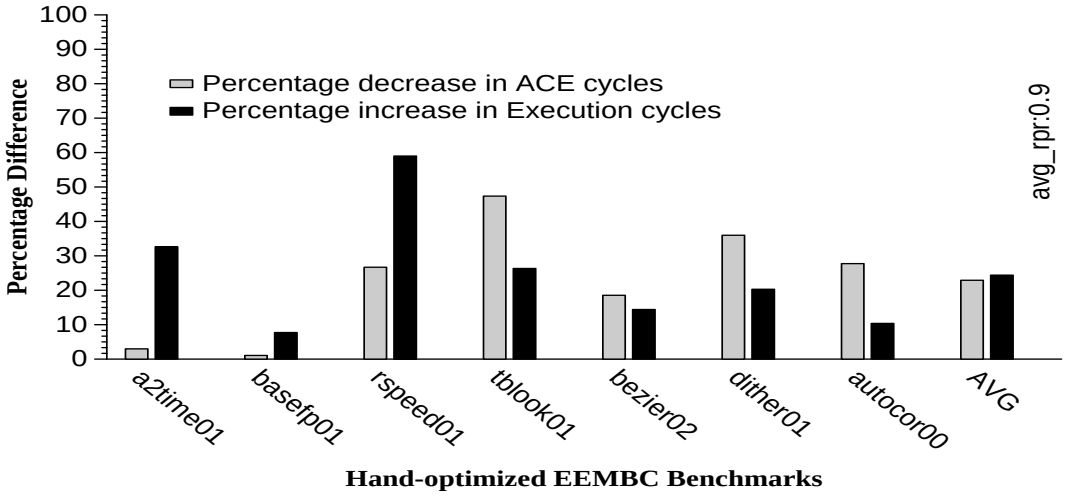

Figure 7.25: Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with fetch-on-demand and dynamic speculation control. Graph shows the potential benefits of AVF throttling.

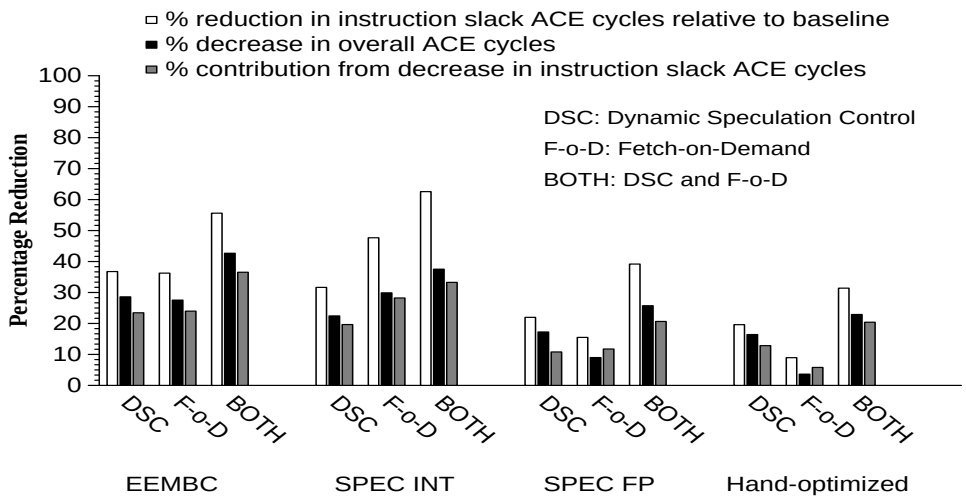

Figure 7.26: For each technique, the first bar shows the reduction in instruction window slack ACE cycles relative to the baseline slack ACE cycles in the instruction window, the second bar shows the overall reduction in ACE cycles, and the third bar shows the contribution of just the reduction in instruction slack ACE cycles in the instruction window to the reduction in total ACE cycles. Graph demonstrates that the techniques are very effective in reducing the slack ACE cycles in the distributed instruction window

each benchmark category it shows the reduction in ACE cycles for all the three techniques. Further, for each technique it presents three bars. The first bar shows the percentage reduction in the instruction window slack ACE cycles relative to the baseline instruction window slack ACE cycles, isolating the effectiveness of the technique in reducing slack ACE cycles in the instruction window. The second bar shows the overall reduction in ACE cycles as before, and finally the third bar shows the contribution of just the reduction in the slack ACE cycles in the instruction window to the overall reduction in ACE cycles.

The first bar shows that all the techniques are able to significantly reduce the instruction window slack ACE cycles from the baseline. Combining both the techniques achieves 55% reduction for EEMBC benchmarks in instruction window slack ACE cycles relative to the baseline slack ACE cycles in the instruction window, 62% reduction for SPEC integer benchmarks, almost 40% reduction for the SPEC floating point benchmarks, and 31% reduction for the hand-optimized benchmarks. The second and third bars together

clearly show that, in all cases, the reduction in instruction window slack ACE cycles contributes a significant fraction to the reduction in total ACE cycles. In fact, the contribution of the instruction window exceeds the overall reduction in ACE cycles in the case of fetch-on-demand for SPEC floating point and hand-optimized benchmarks. For some of these benchmarks, while there is a reduction in the instruction window ACE cycles, some of the other processor structures exhibit an increase in ACE cycles, thus reducing the overall benefit. Overall, this analysis proves that the techniques are effective in reducing the slack ACE cycles in the instruction window, and are thus able to significantly reduce the overall ACE cycles to improve reliability.

7.2.5 Using Reliability Performance Ratio to Tune AVF Throttling

As described in Chapter 6, RPR provides the foundation for a systematic approach where designers work to achieve the reliability target, at the same time optimizing the RPR to achieve low overhead. The optimal RPR is not the same for all designs as it depends on the relative importance of performance and reliability for the application and market segment. RPR can be used as an input parameter to AVF throttling to tune its performance-reliability tradeoff. While it is equally applicable to both dynamic speculation control and fetch-on-demand, we present an example experiment using dynamic speculation control to show the value of RPR in tuning the performance-reliability tradeoff.

We use the input RPR specification to the adaptive algorithm to tune dynamic speculation control to two possible scenarios, and Table 7.1 presents the average results for all the benchmark categories. While an RPR of 1 represents a design that considers reliability and performance to be equally important, an RPR of 4 models a system that is very performance-centric and will accept a performance overhead of 1% only for a reliability improvement of at least 4%. A system with an RPR of 1 is analogous to a system optimized for energy-delay, and an RPR of 4 corresponds to a system optimized for energy-delay^4.

We make three key observations based on an analysis of the EEMBC results. First,

Dynamic Speculation Control	Input RPR	ΔEXEC% Cycles	ΔACE% Cycles	Actual RPR
EEMBC	1	8.2%	28.6%	3.5
	4	3.3%	18.6%	5.6
SPEC INT	1	15.0%	22.5%	1.5
	4	6.7%	16.6%	2.5
SPEC FP	1	15.0%	17.25%	1.2
	4	6.1%	11.0%	1.8
Hand-optimized	1	7.4%	16.4%	2.2
	4	6.6%	13.5%	2.1

Table 7.1: The table shows that RPR can be effectively used to regulate the reliability improvement and performance overhead of dynamic speculation control to meet the input RPR specification.

the actual RPR is substantially larger than the input RPR specification. This is consistent with the goal of the adaptive algorithm that tries to maximize RPR, and is an indicator of the degree of slack available in the program execution that can be exploited by using this specific technique. Second, the actual RPR for an input RPR specification of 4 is substantially greater than for an input RPR of 1, clearly demonstrating the effectiveness of the RPR metric in being able to adapt to specified design requirements. Finally, as expected the absolute reduction in ACE cycles achieved with RPR 4 is lower than with RPR 1 since the algorithm must meet a tighter constraint.

The results for SPEC integer benchmarks are similar, except that dynamic speculation control is unable to meet the input RPR specification of 4, suggesting that we need more efficient techniques such as fetch-on-demand. As mentioned, dynamic speculation control examines four different instruction window sizes for each program interval and chooses the one that meets the input RPR specification, and maximizes the actual RPR. This sampling process incurs performance overhead, and the technique reverts to the baseline window capacity if none of the sizes are able to meet the RPR requirement. The RPR is 2.5 because while some benchmarks (or benchmark phases) are able to meet this RPR target of 4, there are other programs (or program phases) where the adaptive algorithm incurs the performance overhead of sampling but finally chooses to use the baseline instruction window

capacity since none of the other settings satisfy the RPR constraint. This behavior is even more evident for the SPEC floating point and hand-optimized benchmarks where the results basically reflect the sampling overhead of the algorithm when the input RPR specification is equal to 4.

7.3 Selective Redundant Execution (SRE)

Full redundant execution, which redundantly executes all the instructions, and AVF throttling, which completely eliminates redundant execution and instead attempts to carefully reduce the amount of vulnerable state are at two ends of the spectrum of redundant execution. The goal of selective redundant execution (SRE) is to achieve higher reliability than AVF throttling by using redundant execution, while improving the performance overhead over FRE by using redundant execution selectively. Further, our design of selective redundant execution is built upon the previously proposed AVF throttling technique, dynamic speculation control.

We use dynamic speculation control, described in Section 7.2.1, to trigger redundant execution during periods of low single-thread performance [119]. Redundant execution is triggered for those program phases for which the instruction window size, and hence number of speculative blocks, determined by dynamic speculation control falls below a certain threshold. The algorithm takes this as an indication of low single-thread performance, and the potential existence of idle resources that can be cost-effectively used by redundant execution. The advantage of using dynamic speculation control is that even if the number of speculative blocks does not fall below the threshold for redundant execution, there is still some reduction in ACE cycles from just dynamic speculation control. The decision to trigger redundant execution is coupled with the sampling phase of dynamic speculation control, which is repeated every 20 intervals as described in Section 7.2.1. We set the threshold for redundant execution at a speculation depth of 4 TRIPS blocks, which is half the maximum number of TRIPS blocks allowed during normal execution. Figure 7.27 illus-

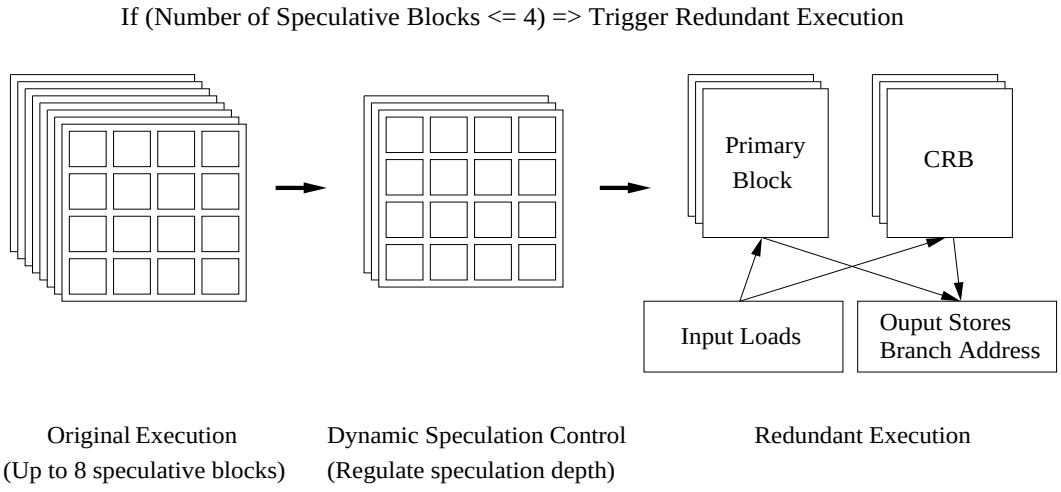

Figure 7.27: Dynamic speculation control based selective redundant execution

trates this two-step mechanism. While we set the threshold at four speculative blocks, the performance-reliability tradeoff of this technique can be varied by appropriately configuring this threshold.

Figures 7.28, 7.29, 7.30, and 7.31 present the results for the SPEC integer, SPEC floating point, EEMBC, and the hand-optimized benchmarks respectively. The technique is very effective for all the benchmark categories. SPEC integer benchmarks exhibit 40.7% reduction in ACE cycles at 10.4% execution time overhead, SPEC floating point benchmarks achieve 26.7% reduction in ACE cycles at 10.2% overhead in execution time, EEMBC benchmarks on the average show 58.7% reduction in ACE cycles at 13.7% execution time overhead, and hand-optimized benchmarks achieve 30% reduction in ACE cycles at 13% overhead. More analysis reveals that SRE is able to achieve substantial improvement in performance overhead because it eliminates between 40-55% of the redundant instructions in comparison with full redundant execution across the different benchmark categories. This is consistent with other approaches for SRE which also eliminate between 30-50% of redundant instructions and achieve comparable improvement in performance overhead [117,118]. Overall SRE achieves significant reliability improvement at reasonable overhead. Further,

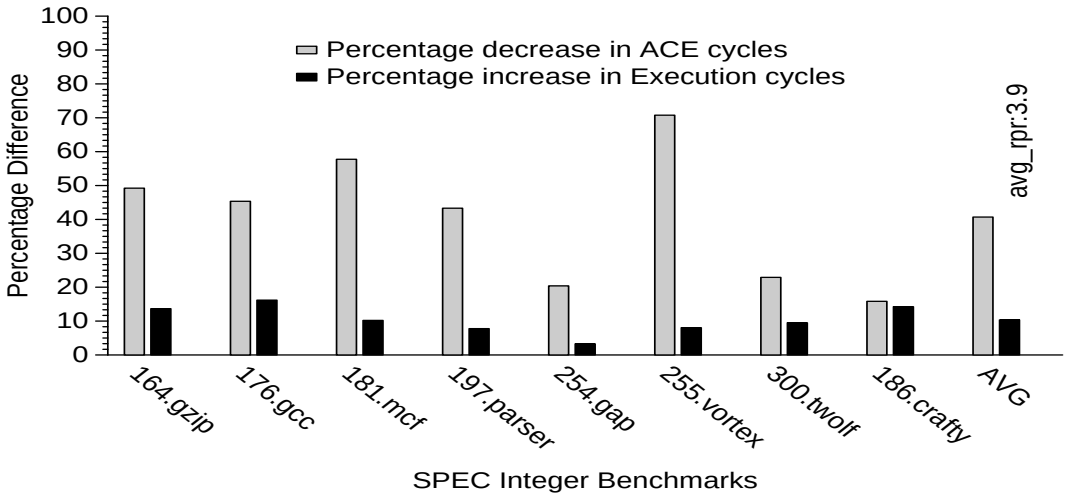

Figure 7.28: Impact on execution and ACE cycles for SPEC integer benchmarks with selective redundant execution.

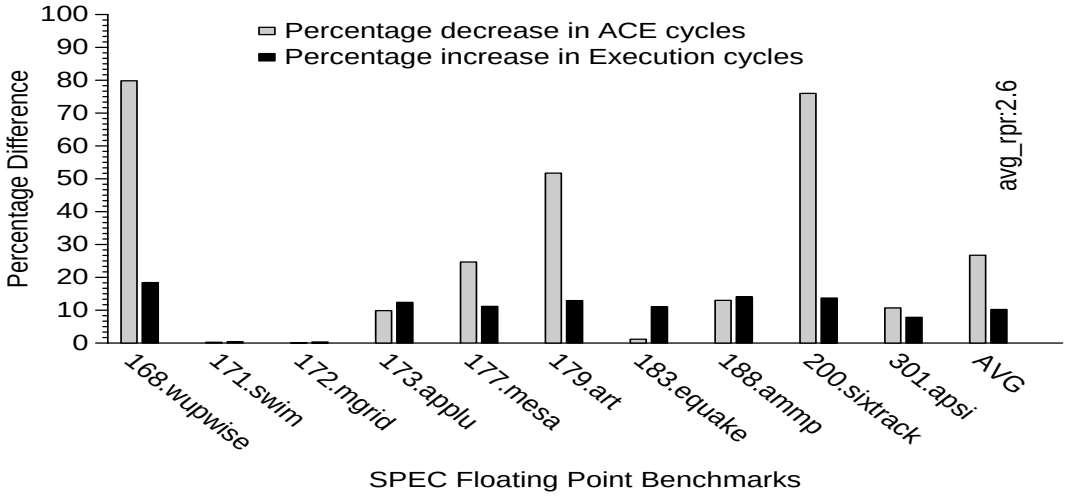

Figure 7.29: Impact on execution and ACE cycles for SPEC floating point benchmarks with selective redundant execution.

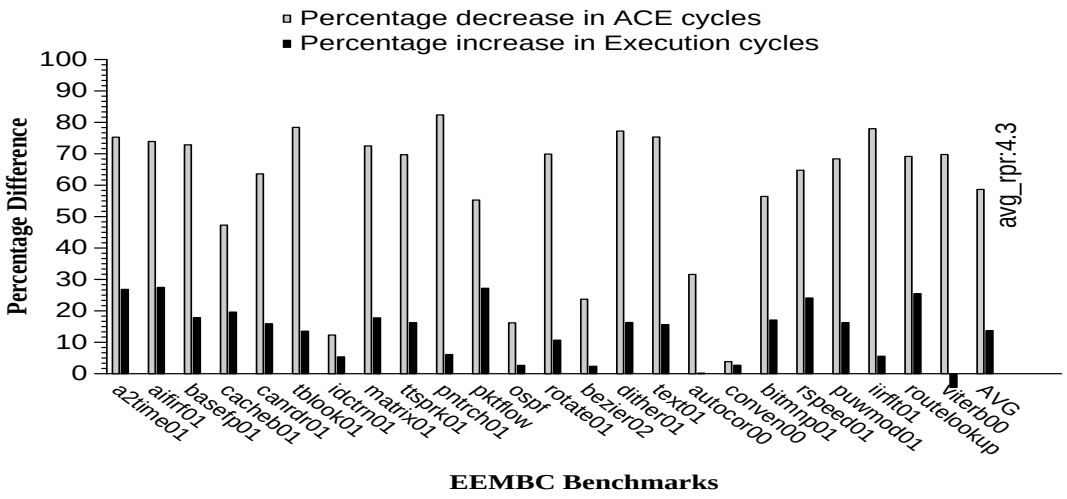

Figure 7.30: Impact on execution and ACE cycles for EEMBC benchmarks with selective redundant execution.

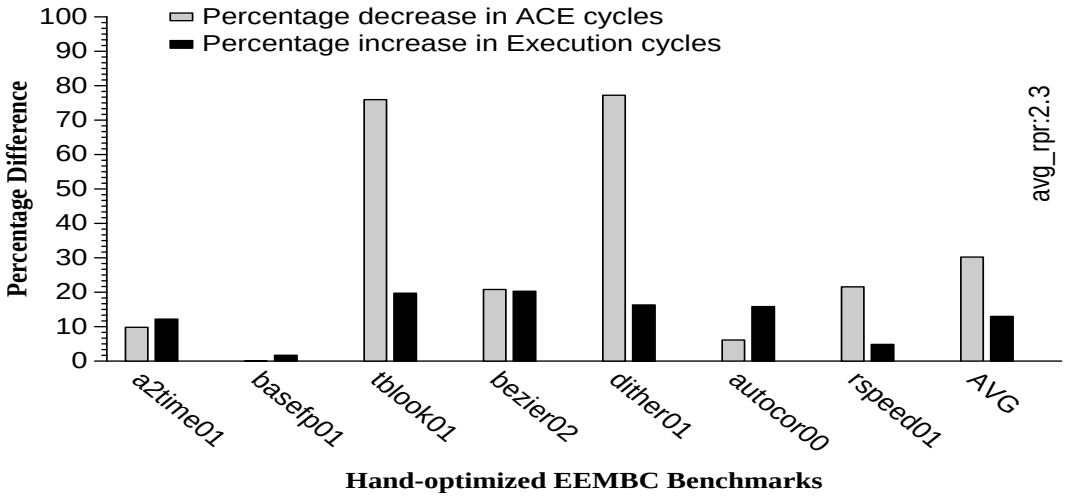

Figure 7.31: Impact on execution and ACE cycles for hand-optimized EEMBC benchmarks with selective redundant execution.

it adds very little complexity to full redundant execution since it reuses the dynamic speculation control framework. Some prior hardware-based approaches have explored other opportunities for skipping redundant execution, but require dedicated structures for measuring the confidence of branch and value prediction and add more complexity [118].

7.4 Summary

This chapter evaluated the performance-reliability tradeoff of techniques from three soft error reliability regimes. In this section, we summarize the insights gained from the evaluation. We not only compare the performance-reliability tradeoff of the different regimes, but also comment on the overheads in power consumption, area, and complexity which separate these different regimes. Table 7.2 presents the results of this comparison. While we perform this analysis at the architectural level, Mitra et al. performed a similar analysis for circuit-level techniques [150]. Finally, we end with a discussion on how these techniques can be applied in conventional architectures.

Performance-Reliability tradeoff: Figure 7.32 shows the comparison of the performance-reliability tradeoff of the three regimes: full redundant execution (FRE), selective redundant execution (SRE), and AVF throttling. The AVF throttling results are basically the results from using the combination of dynamic speculation control and fetch-on-demand presented in Section 7.2.3. For all the benchmark categories, FRE achieves the maximum reduction in ACE cycles followed by SRE, followed by AVF throttling. However, the reduction in ACE cycles achieved by AVF throttling is competitive with SRE. The scenario for performance overhead is slightly different. While SRE and AVF throttling significantly improve the performance overhead over FRE, AVF throttling uniformly incurs slightly greater performance overhead than SRE leading to the smaller RPR in all the benchmark categories (eg: 2.5 vs 4.3 for EEMBC benchmarks). While the RPR of SRE is greater than FRE for EEMBC, SPEC floating point, and hand-optimized benchmarks, it is the opposite for

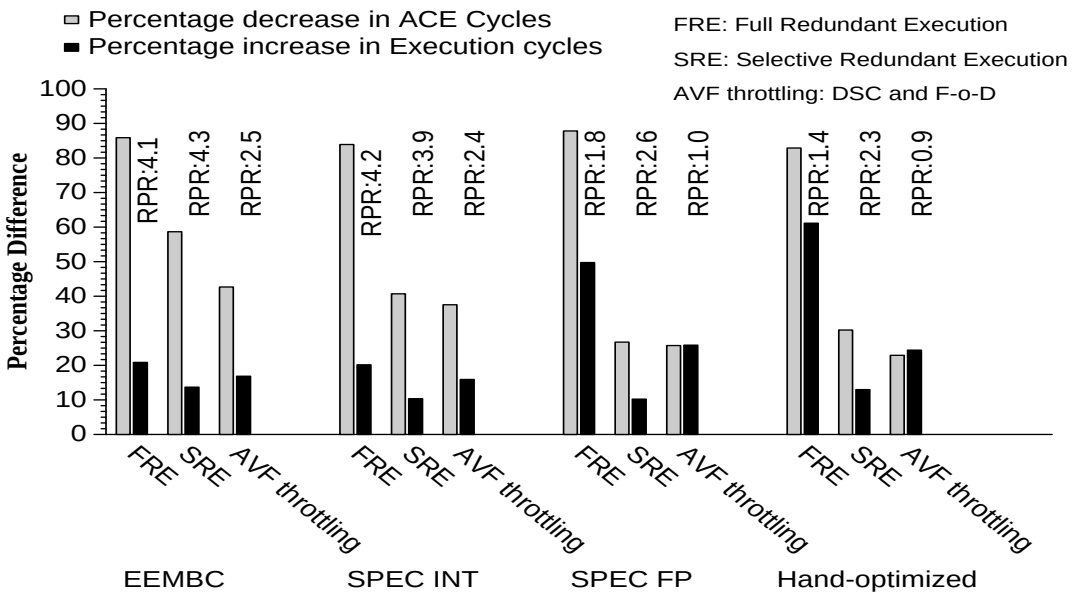

Figure 7.32: Comparison of the three soft error reliability regimes: full redundant execution (FRE), selective redundant execution (SRE), and AVF throttling. Graph shows that the reliability improvement from AVF throttling is comparable to selective redundant execution.

SPEC integer benchmarks, because they incur lower overhead with FRE due to their lower inherent parallelism.

Table 7.2 presents the percentage reduction in ACE cycles and the percentage increase in execution cycles for the three regimes averaged across all the benchmark categories. The results for the compiler-generated EEMBC and SPEC benchmarks, and the hand-optimized EEMBC benchmarks are listed separately. It also presents these results for error correcting codes. Parity adds fault detection capability to a structure, and without adding any extra ACE cycles converts the already exisiting SDC ACE cycles into DUE ACE cycles. As mentioned, since DUE ACE cycles are at least 40X less malicious than SDC ACE cycles, parity achieves a 97.5% reduction in ACE cycles. Since ECC also adds recovery, it improves upon parity and achieves near 100% reduction in ACE cycles for the specific structure. However, as discussed before, adding parity or ECC to timing and performance-critical structures on the datapath can be quite complex.

Regime	**ECC, Parity**	**FRE**	**SRE**	**AVF Throttling**
Description	Incrementally add ECC, Parity to on-chip structures	Full Redundant Execution	Selective Redundant Execution	Reduce Vulnerable State
Percentage decrease in ACE cycles	Near 100% for structures that are protected Parity: $\simeq$ 97.5% ECC: $\simeq$ 100%	Compiler-generated benchmarks 85.9%	Compiler-generated benchmarks 47.6%	Compiler-generated benchmarks DSC: 24.7% F-o-D: 23.6% BOTH: 37.6%
		Hand-optimized benchmarks 82.8%	Hand-optimized benchmarks 30.2%	Hand-optimized benchmarks DSC: 16.4% F-o-D: 3.6% BOTH: 22.9%
Percentage increase in Execution cycles	8-10 FO4 gate delays	Compiler-generated benchmarks 27.6% RPR: 3.1	Compiler-generated benchmarks 12.2% RPR: 3.9	Compiler-generated benchmarks DSC: 11.1% F-o-D 8.0% BOTH: 19.6% RPR: 1.9
		Hand-optimized benchmarks 61.1% RPR: 1.4	Hand-optimized benchmarks 12.9% RPR: 2.3	Hand-optimized benchmarks DSC: 7.4% F-o-D: 15.8% BOTH: 24.4% RPR: 0.9
Power Consumption Overhead	Low	High $\simeq$ 2X	Medium	Low
Area Overhead	11-13% of storage area	$\leq$ 5% (BOQ, LVQ, Store Buffer)	$\leq$ 5% (BOQ, comparison logic, performance counters etc.)	$\leq$ 5% (Performance counters etc.)
Complexity Overhead	Low	High	High	Low
Reference	[21], [2]	[19]	[119], [118], [117]	[2]

Table 7.2: Comparison of the reliability improvement and design tradeoffs between the different regimes

The average results for the other three regimes basically agree with the conclusions above for the individual benchmark categories. All the three regimes, FRE, SRE, and AVF throttling, shows less reliability improvement with more performance overhead for the hand-optimized benchmarks. This is not surprising as benchmarks which have a lot of parallelism and which utilize the processor resources efficiently will experience greater overhead. Prior research that investigated fault tolerance techniques for highly parallel scientific benchmarks in a streaming supercomputer arrived at a similar conclusion [170]. Further, since the TRIPS compiler is still in development, it will be interesting to re-evaluate the gap in performance and the gap in the benefits of these reliability techniques between compiler-generated and hand-optimized benchmarks as the compiler matures.

The scaling analysis in Section 6.1 presented four scaling scenarios each requiring a different reduction in AVF every generation from architectural reliability techniques: a) scenario 1: technology scaling, needing 50% AVF reduction, b) scenario 2: device/circuit techniques, needing 25% AVF reduction, c) scenario 3: error correcting codes, needing 38% AVF reduction, and d) scenario 4: multi-layer reliability, needing 5% AVF reduction. These results corroborate the scaling analysis as they clearly show that the three regimes represent different points on the performance-reliability tradeoff spectrum. Further, these results also suggest that the techniques are appropriate for the different scenarios in the scaling analysis. While FRE is required in Scenario 1, and SRE may be needed in Scenario 3, AVF throttling techniques may be applicable in Scenario 2, and are certainly applicable in Scenario 4. Tackling soft error reliability at multiple layers of the design using device and circuit techniques, and error correcting codes in on-chip structures, allows us to employ AVF throttling techniques which provide reliability improvement competitive with SRE. Although the performance overhead of AVF throttling is slightly greater than SRE, they have many important design advantages over FRE and SRE that are explained below.

Power consumption: FRE incurs at least 2X overhead in power consumption because it redundantly executes all instructions. Recent research has explored combining dynamic

Performance Technique	EPI Throttling Technique	Reference
Scheduling	Low power scheduling	[172]
Speculation	Adaptive processing, Pipeline gating	[133], [173]
Multiprocessing	Asymmetric cores	[174], [105]

Table 7.3: Synergy between AVF and EPI throttling techniques

voltage and frequency scaling (DVFS) with other microarchitectural techniques to reduce the power consumption of redundant execution [128,171], albeit at slightly higher complexity. Since SRE reduces the extent of redundant execution it significantly lowers the power consumption overhead, and can further be combined with the above techniques.

While FRE and SRE increase the power consumption, AVF throttling can potentially reduce the baseline power consumption. We make the important observation that the core mechanisms of many microarchitectural techniques proposed in prior research for improving the energy-per-instruction (EPI) of processors, have important synergies with the AVF throttling mechanisms proposed in this dissertation. Thus these class of techniques can reduce both the power consumption and the vulnerability to soft errors. Similar to AVF throttling that trades concurrency for improving reliability, Table 7.3 presents some EPI throttling mechanisms that trade concurrency to reduce power consumption. Schedulers exploit their knowledge of static instruction slack to maximize concurrency. While power-aware scheduling policies trade concurrency for power consumption [172], the algorithms can be extended to exploit static slack information to reduce the ACE cycles. Adaptive processing [133], and pipeline gating [173] which reduce energy consumption by tuning the processor resources to application needs, effectively also eliminate unnecessary but error prone processor state. Asymmetric cores in a CMP, which perform this tuning in a more coarse-grained fashion, have been shown to be more effective in reducing power consumption by minimizing unnecessary state and computation [105].

Area overhead: As described in Section 6.4.1, the overhead of adding ECC to storage structures is between 11-13% of the array area. Simultaneous multithreading provides an opportunity for eliminating fixed area overhead by adding reconfiguration logic that allows the processor to either operate in normal mode, or in reliable mode where the two threads are dynamically coupled to form a redundant execution pair [19,127]. The area overhead of these and other similar redundant execution techniques are due to the synchronizing queues for input replication and output comparison (described in Section 7.1) all of which occupy less than 5% of the processor area. In addition to these queues, the SRE mechanism evaluated in this chapter also requires modest extra state for performance counters to measure the ACE and execution cycles to implement dynamic speculation control. The AVF throttling techniques further economize on the hardware support to reduce area overhead. Dynamic speculation control only needs performance counters and some control logic for throttling speculation, and fetch-on-demand places almost the entire burden on the scheduler.

Complexity overhead: The synergy between AVF and EPI throttling promises an economy of mechanisms, and is critical in reducing the implementation, and verification complexity of AVF throttling. Section 6.4.1 discussed prior work in extending parity protection to filter false positives due to errors that will eventually be masked [2], and concluded that while a subset of them including ex-ACE and mis-speculated state can be detected using modest support at instruction issue and commit time, catching false positives in predicated-false and dynamically dead state can be fairly complicated and requires dedicated hardware support. By demonstrating the potential of AVF throttling to improve reliability, we corroborate the observation in [2] that designers can take advantage of the complementary nature of AVF throttling that reduces vulnerability, and parity protection that implements detection to improve the overall reliability. Further, we propose that the effectiveness of AVF throttling provides an opportunity to limit the complexity of extended parity protection to a subset of the false positives that do not need the complex hardware support (ex-ace state, and mis-speculated state), and still achieve considerable reliability improvement.

Applicability to conventional architectures: While we use the TRIPS processor for our analysis, FRE and SRE have been investigated in more conventional architectures. Further, speculation throttling and adaptive instruction fetch have also been applied in more conventional processors in prior research with modest modifications to the fetch, decode, and selection logic to improve energy efficiency [173]. While fetch-on-demand operates on compiler generated TRIPS instruction blocks, it can be naturally extended to basic blocks or hyperblocks in conventional processors by using the register rename logic to track the arrival of the register inputs. Alternatively, a compiler based fetch-on-demand scheme has been proposed in prior research for conventional processors to improve performance [161]. In that scheme, a compiler or a profiler is used to annotate each static instruction as critical or deferrable based on the instruction's slack. Deferrable instructions are placed into a deferred queue for later processing, and either use the idle slots in the fetch pipeline or are fetched-on-demand into the processor when their results are needed, similar to what we propose. Fetch-on-demand uses a *reactive* trigger based approach, and is applied at a coarse-grained instruction block granularity. Reactive schemes have also been proposed in earlier work for efficient energy and thermal management [175]. Further, prior research on conventional processors have also explored *predictive* techniques for estimating slack, and applied them at a fine-grained instruction granularity to save energy [159].

Chapter 8

Conclusions

Aggressive technology scaling, rising on-chip integration, and the continued increase in microprocessor power and thermal density threaten both the hard and soft error reliability of future microprocessor designs. Designers must consider processor reliability as a key technology challenge in all market segments, along with performance, and power consumption, and look for efficient solutions at multiple levels of the system design to achieve high reliability at low overhead [4]. While techniques at the device and circuit level have finer control over the susceptibility of individual circuits to errors, architectural solutions are amortized over larger sections of the processor and may have lower overhead. Technology constraints of wire-delay and power consumption, and limits on deep pipelining, have impelled a shift to distributed architectures that rely on modularity in design, and on-chip interconnection networks for communication, and place a greater burden on software for exploiting concurrency from the application to achieve high performance on the distributed substrate [1]. In this dissertation, we make the key observation that these underlying principles of distributed architectures have important synergies that can be exploited to improve processor hard and soft error reliability at low overhead.

8.1 Hard Error Reliability

In this dissertation, we show that the random defect limited yield of chip multiprocessor architectures reduces significantly from 85% at 250nm to 60% at 50nm, suggesting that just having array redundancy in the caches will be insufficient at future technologies to provide us the maximum yield we can hope to achieve at that technology generation. The key idea we propose is to exploit the abundant inherent microarchitectural redundancy in modern and future processor architectures, perhaps with some performance degradation, to achieve significant yield improvement. Exploiting microarchitectural redundancy using mechanisms that mostly already exist in dynamic superscalar architectures improves yield to 99.6% at 50nm, with a maximum reduction in performance in any chip of less than 20%. Recognizing that fault reconfiguration in static architectures also requires the defective configuration to be exposed to the software, we propose a novel compiler-assisted yield enhancement technique and evaluated its potential using the TRIPS architecture. Our results show that the fault-aware scheduling heuristics exploit the redundancy in the TRIPS hardware to successfully reschedule the full set of SPEC and EEMBC benchmarks with less than 4% impact on performance.

The three principles of distributed architectures change the philosophy of design for hard error reliability in three significant ways. Traditionally, techniques primarily focused on providing explicit spares for yield enhancement. While traditional techniques similar to array redundancy (redundant rows and columns) will still be applicable in on-chip caches and queues, the majority of processor components in future distributed architectures will have abundant redundancy, obviating the need for explicit spares. The use of well defined multi-hop, routed, on-chip networks will provide inherently more path redundancy than global wires, with the degree of path redundancy depending on the routing function. The challenge here will be to quantify the defect tolerance for each network based on the routing function, and decide on the appropriate methodology for routing around faults using one of three techniques, addition of explicit routers or links, improving the adaptivity of the

hardware routing function, and software-assisted fault reconfiguration. Finally, depending on the extent to which the software is used to exploit concurrency, and the execution-model, future reliability techniques will need to increasingly use software assistance for maintaining the integrity of the execution model in the presence of a variable number of functional and defective components at different granularities. Further, using software assistance may also help achieve lower overhead.

While future chip multiprocessor architectures will contain substantial redundancy at the processor level [140], that can be exploited using a combination of hardware and software techniques [28, 126], we explore two key advancements in this dissertation. First, we exploit intra-processor redundancy that significantly improves yield if the defect density is large, or if the cores that compose the chip are large. Second, we demonstrate how execution resources within a processor can also be virtualized using software to improve yield at low overhead. We perform fault reconfiguration completely in software, using fault-aware instruction scheduling heuristics that take advantage of the block-atomic execution model in the TRIPS architecture to virtualize the processor resources. While traditional purely hardware based schemes and the fault-aware scheduling heuristics are at two extremes, a hybrid approach that takes advantage of the block-atomic execution model but performs fault reconfiguration using on-chip networks with virtual channels and adaptive routing algorithms is worth exploring in the future. Though we focus primarily on improving chip yield, we recognize that many of the techniques discussed will also help in enhancing the graceful degradation of fail-in-place systems.

8.2 Soft Error Reliability

This dissertation quantitatively demonstrates through detailed modeling that soft error rate, especially that of combinational logic, increases substantially at future technologies. This result emphasizes the need for innovative solutions that extend soft error protection to latches, and combinational logic, and achieve the reliability requirement for that particular

system (or market segment) while appropriately balancing the power consumption, area, and complexity overhead.

Prior approaches have examined using the redundancy, and the software infrastructure (eg: threads) available in distributed architectures for improving soft error reliability at low overhead [19, 127, 128]. In this dissertation, we propose a new better-than-worst-case technique called AVF throttling for improving soft error reliability, and demonstrate its applicability and potential for reducing soft error rate. AVF throttling is based on fault avoidance rather than fault detection, and trades concurrency for the amount of processor state vulnerable to soft errors to improve reliability. It is based on the key observation that future architectures that increasingly rely on exploiting concurrency for achieving high performance, may also provide correspondingly greater opportunities for exploiting execution slack to reduce the amount of vulnerable state. We show that in the TRIPS architecture around 90% of the vulnerable state is due to slack, and AVF throttling mechanisms are able to achieve significant reliability improvement comparable to selective redundant execution of 25-40% for a set of EEMBC and SPEC benchmarks. While AVF throttling incurs a slightly higher performance overhead than selective redundant execution, it has the potential for considerably smaller area, power consumption, and complexity overhead when compared to full and selective redundant execution.

Our results indicate that exploiting slack has significant potential to improve soft error reliability. While we limit our analysis to local slack, we expect that accounting for global slack will provide even greater benefits and is worth exploring in the future. Our current methodology integrates estimation of *Min*, *Contention*, and *Local Slack* ACE cycles into the ACE analysis framework, for estimating reliability and studying the interaction between performance and reliability early in the design cycle. In the future, we believe that the critical path analysis framework can be elegantly extended to perform ACE analysis, and compute the *Min*, *Contention*, and *Slack* (both local and global) ACE cycles [159, 168, 169]. This comprehensive unified framework will enable architects to study the tradeoff between

concurrency and slack, and its impact on soft error reliability in greater detail.

In this dissertation, we explore two complementary techniques for exploiting slack in the TRIPS architecture, dynamic speculation control at the inter-block granularity, and fetch-on-demand at the intra-block level. We show that fetch-on-demand which exploits a more detailed knowledge of the TRIPS microarchitecture to achieve finer control over slack, in some cases provides greater reduction in slack than dynamic speculation control that operates at a higher level. We conclude that a similar approach can be used to arrive at efficient techniques for exploiting slack in other processor structures and architectures also. Of course, the degree of slack available depends both on the application and the architecture, and well-tuned specialized architectures for applications with abundant parallelism may not contain much slack to begin with [170].

While prior techniques proposed for exploiting slack are quite complex [159], *Fetch-on-Demand* uses two key techniques to reduce complexity without sacrificing the benefits: a) slack estimation at the block granularity that provides a favorable compromise between complexity and timeliness, and b) a hybrid approach using the compiler for slack estimation, and the hardware for exploiting slack based on information from the compiler. The design space for estimating and exploiting slack warrants more investigation in the future. Further, sophisticated techniques using software and hardware cooperation for exploiting slack at coarse-grained block-, function-, or transaction-level can form the building block for achieving multiple technology objectives including reliability, and power efficiency.

8.3 Final Thoughts

Due to limitations in further scaling of clock frequency, power-efficient exploitation of concurrency is presently the primary design constraint for achieving performance gains in computer systems. However, in this dissertation we demonstrate that hard and soft error reliability are key emerging technology challenges that must be considered to maintain system reliability at acceptable levels. In the future, analysis of the performance-reliability trade-

off will be an important component of the design process along with studying the power consumption, area, and complexity tradeoffs.

We demonstrate that the principles of distributed architectures provide a solid foundation for improving hard error reliability at low overhead. The reliability enhancement techniques proposed successfully take into consideration both the opportunities and the extra demands placed by important new features of future distributed architectures including on-chip interconnection networks, and the greater reliance on software, to achieve significant improvement in reliability at low overhead.

Performing performance-power consumption tradeoff analysis has become a standard component of mainstream processor design, and no more can microarchitectural widgets be included in a processor based on the performance improvement alone. In this dissertation, we propose Reliability Performance Ratio (RPR), a new metric for performing similar analysis of the performance-reliability tradeoff which will become a critical component in the future. Chapter 6 provides a concrete example in the TRIPS architecture where the larger capacity of the instruction window and the register operand storage hierarchy increase their soft error vulnerability by 2-4X over conventional architectures. Based on prior research that demonstrates average performance improvements of 3X on hand-optimized benchmarks, this results in an RPR of approximately 1 for this design decision.

At a high level, power consumption is a function of the number of transistors, and is a problem that approximately scales with the Moore's law increase in on-chip integration. However, there is no such silver bullet like full redundant execution for solving this problem, and power consumption and the performance-power consumption tradeoff have been successfully tackled using configurable mechanisms at multiple levels of the design from the device level all the way up to the architecture and the software. Due to the constant or even increasing soft error rate(SER)-per-individual circuit shown in this dissertation, soft error reliability will also become a problem of Moore's law proportions at the transistor level. However, the critical difference from power consumption is that circuit, microarchi-

tecture, architecture, and program-level masking factors can significantly reduce this raw vulnerability. In spite of this, history points towards adopting full redundant execution for solving this problem, and we argue that designing flexible and configurable mechanisms will be a key requirement to achieve low overhead. While selective redundant execution add this configurability it still comes at the cost of higher complexity and power consumption. This emphasizes the need for innovative techniques such as AVF throttling which have the potential to improve reliability, power consumption, and complexity.

Bibliography

[1] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[2] Christopher Weaver, Joel S. Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 264–275, June 2004.

[3] International technology roadmap for semiconductors 2006 update: Process integration, devices, and structures. `http://www.itrs.net`.

[4] Critical reliability challenges for the international technology roadmap for semiconductors. Technical Report 03024377A-TR, International Sematech Technology Transfer, 2003.

[5] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, June 2004.

[6] Shekhar Borkar. VLSI design challenges for gigascale integration. In *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, page 27, 2005.

[7] Walter L. Heimerdinger and Chuck B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, Pittsburgh, PA, 1992.

[8] Scott Hareland, Jose Maiz, Mohsen Alavi, Kaizad Mistry, Steve Walsta, and Changhong Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. *Symposium on VLSI Technology Digest of Technical Papers*, pages 73–74, June 2001.

[9] E. H. Cannon, D. D. Reinhardt, M. S. Gordon, and P. S. Makowenskyj. SRAM SER in 90, 130, and 180nm bulk and SOI technologies. In *IEEE 42nd Annual International Reliability Physics Symposium*, pages 300–304, April 2004.

[10] Kartik Mohanram and Nur A. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Proceedings of the International Test Conference*, pages 893–901, September 2003.

[11] Tanay Karnik, Sriram Vangal, V. Veeramachaneni, Peter Hazucha, Vasantha Erraguntla, and Shekhar Borkar. Selective Node Engineering for Chip-Level Soft Error Rate Improvement. *VLSI Circuits Symposium*, pages 204–205, June 2002.

[12] Krishna Seshan, Timothy J. Maloney, and Kenneth J. Wu. The quality and reliability of intel's quarter micron process. *Intel Technology Journal*, (Q3), 1998.

[13] Nicholas P. Mencinger. A mechanism-based methodology for processor package reliability assessments. *Intel Technology Journal*, (Q3), 2000.

[14] R. V. White and F. M. Miles. Principles of fault tolerance. *Applied Power Electronics Conference and Exposition (APEC)*, 1(3-7):18–25, March 1996.

[15] Nhon T. Quach. High availability and reliability in the Itanium processor. *IEEE Micro*, 20(5):61–69, 2000.

[16] D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd. Fault-tolerant design of the IBM pSeries 690 system using POWER4 processor technology. *IBM Journal of Research and Development*, 46(1):77, January 2002.

[17] Todd Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, November 1999.

[18] Lisa Spainhower and Thomas A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, September/November 1999.

[19] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[20] T. Calin, R. Velazco, M. Nicolaidis, S. Moss, S. D. LaLumondiere, V. T. Tran, R. Koga, and K. Clark. Topology-related upset mechanisms in design hardened storage cells. In *Fourth European Conference on Radiation and its Effects on Components and Systems (RADECS)*, pages 484–488, September 1997.

[21] Tyler Thorp and Dean Liu. Analysis of blocking dynamic circuits. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, volume 11, pages 744–748, October 2002.

[22] Alexander Irion, Gundolf Kiefer, Harald Vranken, and Hans-Joachim Wunderlich. Circuit partitioning for efficient logic BIST synthesis. In *Design, Automation, and Test in Europe*, pages 86–91, March 2001.

[23] Graham Hetherington, Tony Fryars, Nagesh Tamarapalli, Mark Kassab, Abu Hassan, and Janusz Rajski. Logic BIST for large industrial designs: Real issues and case studies. In *International Test Conference (ITC)*, pages 358–367, January 1999.

[24] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceeding of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73–82, October 2006.

[25] Stratus continous processing technology. *Stratus White Paper*, 2003.

[26] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: Building high availability systems with commodity multi-core processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[27] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division White Paper*, November 1997.

[28] J. Jann, L. M. Browning, and R. S. Burugula. Dynamic reconfiguration: Basic building block for autonomic computing on IBM pSeries servers. *IBM Systems Journal*, 42(1):29–37, March 2003.

[29] Nicholas J. Wang and Sanjay J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 30–39, June 2005.

[30] E.N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[31] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for microarchitectural awareness of lifetime reliability. *In Proceedings of the Annual International Symposium on Computer Architecture*, June 2004.

[32] M. Agarwal, B. Paul, M. Zhang, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *IEEE VLSI Test Symposium*, pages 277–286, 2007.

[33] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[34] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate vs. IPC : The end of the road for conventional microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[35] M. S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th International Symposium of Computer Architecture*, pages 14–24, May 2002.

[36] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, February 2001.

[37] Premkishore Shivakumar, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design*, pages 481–488, October 2003.

[38] The International Technology Roadmap for Semiconductors. Semiconductor Industry Association, 2003.

[39] A. Vassighi, O. Semenov, M. Sachdev, A. Keshavarzi, and C. Hawkins. CMOS IC technology scaling and its impact on burn-in. *IEEE Trans. on Devices and Materials Reliability*, 4(2), June 2004.

[40] Xiaolei Li, A. J. Strojwas, and M. F. Antonelli. Holistic yield improvement methodology. *Semiconductor Fabtech Journal*, 8(7):257–265, July 1998.

[41] Israel Koren and Zahava Koren. Defect tolerant VLSI circuits: Techniques and yield analysis. 86(9):1819–1838, September 1998.

[42] W. Maly and J. Deszczka. Yield estimation model for VLSI artwork evaluation. 19(6):226–227, March 1983.

[43] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Microprocessor Forum presentation, October 1996.

[44] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, August 2001.

[45] S. Gupta, S.W. Keckler, and D.C. Burger. Technology independent area and delay estimations for microprocessor building blocks. Technical Report TR-00-05, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, February 2001.

[46] Subbarao Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, Department of Computer Sciences, University Of Wisconsin Madison, 1998.

[47] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March-April 1999.

[48] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Patnaik, and Josep Torellas. FlexRAM: Towards an Advanced Intelligent Memory System. *International Conference on Computer Design*, pages 192–201, October 1999.

[49] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-dominated on-chip caches. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[50] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings*

of the 34th International Symposium on Microarchitecture, pages 40–51, December 2001.

[51] Kevin Krewell. Intel looks to Core for success. *Microprocessor Report*, March 2006.

[52] Jiri Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pages 42–46, 1997.

[53] Peter Liden, Peter Dahlgren, Rolf Johansson, and Johan Karlsson. On latching probability of particle induced transients in combinational networks. In *Proceedings of the 24th Symposium on Fault-Tolerant Computing*, pages 340–349, 1994.

[54] J. Ziegler. Terrestrial cosmic ray intensities. *IBM Journal of Research and Development*, 42(1):117–139, January 1998.

[55] J. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.

[56] L. B. Freeman. Critical charge calculations for a bipolar SRAM array. *IBM Journal of Research and Development*, 40(1):119–129, January 1996.

[57] K. Johansson, P. Dyreklev, B. Granbom, M.C. Calvet, S. Fourtine, and O. Feuillatre. In-flight and ground testing of single event upset sensitivity in static RAM's. *IEEE Transactions on Nuclear Science*, 45(3):1628–1632, June 1998.

[58] E.L. Peterson, P. Shapiro, J.H. Adams, and E.A. Burke. Calculation of cosmic-ray induced soft upsets and scaling in VLSI devices. *IEEE Transactions on Nuclear Science*, 29(6):2055–2063, December 1982.

[59] J.C. Pickel. Effect of CMOS miniaturization on cosmic-ray-induced error rate. *IEEE Transactions on Nuclear Science*, 29(6):2049–2054, December 1982.

[60] Peter Hazucha and Christer Svensson. Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate. *IEEE Transactions on Nuclear Science*, 47(6):2586–2594, Dec. 2000.

[61] Grant McFarland. *CMOS Technology Scaling and Its impact on cache delay*. PhD thesis, Department of Electrical Engineering, Stanford University, 1997.

[62] Peter Hazucha. *Background Radiation and Soft Errors in CMOS Circuits*. PhD thesis, Linkping Studies in Science and Technology, 2000.

[63] Mark A. Horowitz. Timing Models For MOS Circuits. Technical Report SEL83-003, Integrated Circuits Laboratory, Stanford University, 1983.

[64] M. J. Bellido-Diaz, J. Juan-Chico, A. J. Acosta, M. Valencia, and J.L.Huertas. Logical modelling of delay degradation effect in static CMOS gates. *IEEE Proc-Circuits Devices Syst.*, 147(2):107–117, April 2000.

[65] Steven J.E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[66] Kerry Bernstein. Personal communication.

[67] Premkishore Shivakumar, Michael Kistler, Stephen Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems & Networks*, pages 389–398, June 2002.

[68] Pentium II Processor Specification Update. Intel Corporation.

[69] L. W. Massengill, A. E. Baranski, D. O. Van Nort, J. Meng, and B. L. Bhuva. Analysis of single-event effects in combinational logic – simulation of the AM2901 bitslice processor. *IEEE Trans. on Nuclear Science*, 47(6):2609–2615, December 2000.

[70] Bin Zhang, Wei-Shen Wang, and Michael Orshansky. FASER: Fast analysis of soft error susceptibility for cell-based designs. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 755–760, March 2006.

[71] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Assessing SEU vulnerability via circuit-level timing analysis. In *Proceedings of the 1st Workshop on Architectural Reliability*, November 2005.

[72] Changhong Dai, Nagib Hakim, Scott Hareland, Jose Maiz, and Shiuh-Wuu Lee. Alpha-SER modeling and simulation for sub-0.25um cmos technology. *Symposium on VLSI Technology Digest of Technical Papers*, pages 81–82, 1999.

[73] Bijan Davari. CMOS Technology Scaling, 0.1um and Beyond. In *International Electron Devices Meeting*, pages 555–558, December 1996.

[74] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz. Radiation-induced soft error rates of advanced CMOS bulk devices. In *IEEE 44th Annual International Reliability Physics Symposium*, pages 217–225, March 2006.

[75] Y. Tosaka, S. Satoh, T. Itakura, H. Ehara, T. Ueda, G.A. Woffinden, and S.A. Wender. Measurement and analysis of neutron-induced soft errors in sub-half-micron circuits. *IEEE Transactions on Electron Devices*, 45(7):1453–1458, July 1998.

[76] Y. Tosaka, S. Satoh, K. Suzuki, T. Sugii, H. Ehara, G.A. Woffinden, and S.A. Wender. Impact of cosmic ray neutron induced soft errors on advanced submicron CMOS circuits. *Symposium on VLSI Technology Digest of Technical Papers*, pages 148–149, June 1996.

[77] H. Cha and J. H. Patel. A logic-level model for α-particle hits in CMOS circuits. In *International Conference on Computer Design*, pages 538–542, October 1993.

[78] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron CMOS logic circuits. *IEEE Journal of Solid State Circuits*, 30(7):830–834, July 1995.

[79] Y.Tosaka, H.Kanata, S.Satoh, and T.Itakura. Simple method for estimating neutron-induced soft error rates based on modified BGR method. *IEEE Elec. Dev. Lett.*, 20(2):89–91, Feb 1999.

[80] P. C. Murley and G. R. Srinivasan. Soft-error monte carlo modeling program, SEMM. *IBM Journal of Research and Development*, 40(1):109–118, January 1996.

[81] M. Baze and S. Buchner. Attenuation of single event induced pulses in CMOS combinational logic. *IEEE Trans. on Nuclear Science*, 44(6):2217–2223, December 1997.

[82] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. Comparison of error rates in combinational and sequential logic. *IEEE Transactions on Nuclear Science*, 44(6):2209–2216, December 1997.

[83] N. Seifert and N. Tam. Timing Vulnerability Factors of Sequentials. *IEEE Trans. on Devices and Materials Reliability*, 4(3):516–522, September 2004.

[84] Feng Wang, Yuan Xie, R. Rajaraman, and B. Vaidyanathan. Soft error rate analysis for combinational logic using an accurate electrical masking model. *Proceedings of the International Conference on VLSI Design*, pages 165–170, January 2007.

[85] Ming Zhang and Naresh R. Shanbhag. A Soft Error Rate Analysis (SERA) Methodology. *International Conference on Computer Aided Design*, pages 111–118, November 2004.

[86] Rajeev R. Rao, Kaviraj Chopra, David Blaauw, and Dennis Sylvester. An efficient static algorithm for computing the soft error rates of combinational circuits. In *Proceedings of the conference on Design, automation and test in Europe*, pages 164–169, March 2006.

[87] Normal Seifert, David Moyer, Norman Leland, and Ray Hokinson. Historical Trend in Alpha-Particle induced Soft Error Rates of the Alpha(TM) Microprocessor. In *IEEE 39th Annual International Reliability Physics Symposium*, pages 259–265, May 2001.

[88] Tanay Karnik, Bradley Bloechel, K. Soumyanath, Vivek De, and Shekhar Borkar. Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches Beyond 0.18um. *Symposium on VLSI Circuits*, pages 61–62, June 2001.

[89] J. Blome, S. Mahlke, D. Bradley, and K. Flautner. A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor. In *Proceedings of the First Workshop on Architecture Reliability*, November 2005.

[90] Eric Rotenberg. AR/SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *Proceedings of the 29th Annual International Symposium on Fault Tolerant Computing*, pages 84–91, June 1999.

[91] Steven K Reinhardt and Shubhendu Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, July 2000.

[92] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendhu S. Mukherjee. Design and evaluation of hybrid fault detection systems. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[93] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, December 1997.

[94] Joseph Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 140–150, June 1983.

[95] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[96] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Comput er Architecture*, pages 282–293, June 2000.

[97] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[98] Stephen W. Keckler, Doug Burger, Charles R. Moore, Ramadass Nagarajan, Karthikeyan Sankaralingam, Vikas Agarwal, M.S. Hrishikesh, Nitya Ranganathan, and Premkishore Shivakumar. A wire-delay scalable microprocessor architecture for high-performance systems. In *Proceedings of the 2003 International Solid-State Circuits Conference*, volume 1, pages 168–169, February 2003.

[99] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 291–302, December 2003.

[100] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Walter Lee Jae-Wook Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The RAW microprocessor: A compu-

tational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[101] J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–26, January 2001.

[102] Poonacha Kongetira, Karthirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[103] Kevin Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[104] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R . Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July 2005.

[105] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, December 2003.

[106] Karthikeyan Sankaralingam. *Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities*. PhD thesis, Department of Computer Sciences, University Of Texas at Austin, 2006.

[107] Changkyu Kim, Simha Sethumadhavan, Nitya Ranganathan, Haiming Liu, Doug Burger, and Stephen W. Keckler. Elastic threads on composable processors. Technical Report TR-06-09, Department of Computer Sciences, University of Texas at Austin, 2006.

[108] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, Madhu Saravana Sibi Govindan, Paul Gratz, Divya Gulati,

Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethmadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 480–491, December 2006.

[109] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[110] Paul Gratz, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and evaluation of a dynamically routed processor operand network. In *Proceedings of the 1st International Symposium on Networks-on-Chip*, May 2007.

[111] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen Keckler, and Charles Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[112] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[113] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

[114] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[115] Katherine Coons, Xia Chen, Sundeep Kushwaha, Kathryn McKinley, and Doug Burger. A spatial path scheduling algorithm for edge architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[116] Chris Feige and M.J. Geuzebroek. Logic BIST technology evaluation: an industrial case study. In *European Test Workshop*, May 2001.

[117] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendhu S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Transactions on Architecture and Code optimization*, 2(4):366–396, September 2005.

[118] Angshuman Parashar, Sudhanva Gurumurthi, and Anand Sivasubramaniam. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 95–105, October 2006.

[119] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. *IEEE Micro*, 26(1):92–99, 2006.

[120] Chien-Chun Su and Kang G. Shin. Adaptive fault-tolerant deadlock-free routing in meshes and hypercubes. *IEEE Trans. Comput.*, 45(6):666–683, 1996.

[121] Daniel H. Linder and Jim C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Trans. Comput.*, 40(1):2–12, 1991.

[122] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high performance processor pipeline. In *International Conference on Dependable Systems and Networks*, pages 61–70, June 2004.

[123] Shubhendu S. Mukherjee, Christopher Weaver, Joel S. Emer, Steven K. Reinhardt, and Todd M. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 29–40, December 2003.

[124] Jonathan Chang, George A. Reis, Neil Vachharajani, Ram Rangan, and David I. August. Non-uniform fault tolerance. In *Proceedings of the 2nd Workshop on Architectural Reliability*, December 2006.

[125] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *In Proceedings of the Annual International Symposium on Computer Architecture*, pages 87–98, 2002.

[126] Russ Joseph. Exploring salvage techniques for multi-core architectures. In *Proceedings of the 2nd Workshop on High Performance Computing Reliability Issues*, February 2006.

[127] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks*, June 2007.

[128] M. Wasiur Rashid, Edwin J. Tan, Michael C. Huang, and David H. Albonesi. Power-efficient error tolerance in chip multiprocessors. *IEEE Micro*, 25(6):60–70, 2005.

[129] Kevin Krewell. Marketing PC performance. Microprocessor Report, November 2001.

[130] J. A. Farrell and T. C. Fischer. Issue logic for a 600MHz Out of Order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[131] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[132] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[133] David H. Albonesi, Rajeev Balasubramonian, Steve Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12):49–58, 2003.

[134] Alexander Klaiber. The technology behind crusoe processors. *Transmeta White Paper*, January 2000.

[135] Jayanth Srinivasan and Sarita V. Adve. Predictive Dynamic Thermal Management for Multimedia Applications. *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 2003)*, pages 109–120, June 2003.

[136] J. F. Meyer. On evaluating the performability of degradable computer systems. *IEEE Transactions on Computers*, 29(8):720–731, August 1980.

[137] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: low-cost adaptive networks for multiprocessors. *J. ACM*, 42(1):91–123, 1995.

[138] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Trans. Parallel Distrib. Syst.*, 4(4):466–475, 1993.

[139] Ethan Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 160–171, 2005.

[140] R. Nair. Effect of increasing chip density on the evolution of computer architectures. *IBM Journal of Research and Development*, 46(2/3):223–234, March 2002.

[141] C. H. Stapper, A. N. McLaren, and M. Dreckmann. Yield model for productivity optimization of VLSI memory chips with redundancy and partially good product. *IBM Journal of Research and Development*, 24(3):398–409, May 1980.

[142] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 51–60, June 2004.

[143] A. DeHon and H. Naeimi. Seven strategies for tolerating highly defective fabrication. *IEEE Design and Test of Computers*, 22(4):306–315, July 2005.

[144] Dennis Sylvester, David Blaauw, and Eric Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design and Test of Computers*, pages 484–490, November 2006.

[145] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Lifetime Reliability: Toward an Architectural Solution. *IEEE Micro: Special Issue on Future Trends in Microarchitecture*, 25(3):70–80, June 2005.

[146] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *8th International Symposium on High-Performance Computer Architecture*, pages 29–40, February 2002.

[147] Anoop Iyer and Diana Marculescu. Microarchitectural level power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):230–239, June 2002.

[148] J. Henessey. The future of systems research. *IEEE Computer*, 32(8):27–33, July 1999.

[149] Tanay Karnik, Peter Hazucha, and Jagdish Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, April 2004.

[150] Subhasish Mitra, Tanay Karnik, Norbert Seifert, and Ming Zhang. Logic soft errors in sub-65nm technologies design and CAD challenges. In *Proceedings of the 42nd annual conference on Design automation*, pages 2–4, June 2005.

[151] H. Peter Hofstee. Power-constrained microprocessor design. In *Proceedings of the 2002 International Conference on Computer Design*, pages 14–16, 2002.

[152] Michael Nicolaidis. Carry checking/parity prediction adders and ALUs. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(1):121–128, February 2003.

[153] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Trans. Dependable Secur. Comput.*, 1(1):87–96, 2004.

[154] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower. IBM z990 soft error detection and recovery. *IEEE Transactions on Devices and Materials Reliability*, 5(3):419–427, September 2005.

[155] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 214–224, December 2001.

[156] Avi Mendelson and Neeraj Suri. Designing high-performance and reliable superscalar architectures: The out of order reliable superscalar (O3RS) approach. *International Conference on Dependable Systems and Networks*, pages 473–481, June 2000.

[157] Subhasish Mitra and Edward J. McCluskey. Which concurrent error detection scheme to choose? In *Proceedings of the 2000 IEEE International Test Conference*, pages 985–994, October 2000.

[158] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[159] Brian A. Fields, Rastislav Bodík, and Mark D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[160] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *International Conference on Dependable Systems and Networks*, pages 434–443, June 2005.

[161] G. Muthler, D. Crowe, S. Patel, and S. Lumetta. Instruction fetch deferral using static slack. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 51–61, December 2002.

[162] Smitha M. Kalappurakkal. Reducing the Soft Error Rate of a High Performance Microprocessor Using Front-End Throttling. Master's thesis, Department of Electrical and Computer Engineering, University of Maryland, 2006.

[163] Ramadass Nagarajan. *Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism*. PhD thesis, Department of Computer Sciences, University Of Texas at Austin, 2007.

[164] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. Examining ACE analysis reliability estimates using fault-injection. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[165] Aaron Smith, Ramadass Nagarajan, Karthikeyan Sankaralingam, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–102, December 2006.

[166] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 148–159, December 1998.

[167] Steven S. Muchnick and Phillip B. Gibbons. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, 39(4):167–174, 2004.

[168] Brian A Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[169] Ramadass Nagarajan, Xia Chen, Robert G. Mcdonald, Doug Burger, and Stephen W. Keckler. Critical path analysis of the TRIPS architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 37–47, March 2006.

[170] Mattan Erez, Nuwan Jayasena, Timothy J. Knight, and William J. Dally. Fault tolerance techniques for the merrimac streaming supercomputer. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 29, 2005.

[171] Niti Madan and Rajeev Balasubramonian. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems*, 18(8), August 2007.

[172] Mark Toburen, Thomas M. Conte, and Matthew Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 1998.

[173] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.

[174] Ed Grochowski, Ronny Ronen, John Paul Shen, and Hong Wang. Best of both latency and throughput. In *Proceedings of the 2004 International Conference on Computer Design*, pages 236–243, October 2004.

[175] M. Huang, J. Renau, S-M. Yoo, and Josep Torrellas. A framework for dynamic energy efficiency and temperature management. In *Proceeding of the 33rd Annual International Symposium on Microarchitecture*, pages 202–213, December 2000.

Vita

Premkishore Shivakumar was born in Madras, India on March 25th 1979, to Dr. Shivakumar Srinivasan and Dr. Kamalam Shivakumar. After graduating from P.S. Senior Secondary High School in 1996, he was admitted to the Indian Institute of Technology, Madras. He was awarded the Bachelor of Technology in Electrical Engineering in 2000. In the fall of 2000, he was admitted to the doctoral program in the Department of Computer Sciences at the University of Texas at Austin. While pursuing his Ph.D. degree, he received the degree of Master of Science in Computer Sciences in May 2004.

Permanent Address: Flat 2D, Sindur Prestige Point
No.33, First Main Road
Kasthuribha Nagar, Adayar
Chennai, India 600020.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.